TRANSFORMER SPECIFICATION LANGUAGE: A SYSTEM FOR GENERATING ANALYZERS AND ITS APPLICATIONS

by

Junghee Lim

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences Department)

at the

UNIVERSITY OF WISCONSIN–MADISON

2011

To my mom and dad. . .

ACKNOWLEDGMENTS

I am indebted to the many people who made this thesis possible. First and foremost, my deepest gratitude goes to my advisor, Prof. Thomas Reps. This thesis would not have been possible without his guidance. His inspiration, motivation, immense knowledge, and enthusiasm in research have motivated all his advisees, including me. I am especially grateful for his patience and his best efforts to teach me how to be a good researcher, and his ungrudging advice and encouragement to complete my graduate study.

I would also like to thank him for an unforgettable year in Europe when he was taking me along during his sabbatical. The year I spent in Paris was the most interesting time in my life. I am very grateful for his sincere care concerning the fact that I was spending that year separate from my husband.

He has provided continuous help and encouragement in many respects other than research as well. I will never forget the thoughtful flowers he gave me every year remembering my dad. Whenever I was having a hard time in many respects, he was willing to listen to me, stand by me, and give the strength to overcome the adversities. It was one of the luckiest things in my life to have him as my advisor. I could not have imagined having a better advisor and mentor for my graduate study and research.

I would like to dedicate this thesis to my family. Without their constant support and unconditional love, I could not have completed my graduate study. My father never stopped spiritually supporting me from thousands of miles away, even in the moment suffering from his health problem. I feel a heartfelt sadness that I couldn't stay by him at the moment he passed away. He must have been very proud of me. I am thankful for my mother staying healthy and helping me have

peace in my mind. They have been a constant source of love and encouragement and supported me spiritually throughout my life.

Most importantly, I am heartily thankful to my husband, Dr. Min-Sik Kim, for making me laugh when I needed it and for making available his support in a number of ways. I would also like to thank my sister and brother, and my parents and sisters and brother in law for their love and support. My dissertation would have been meaningless without my family.

My sincere thanks also goes to my undergraduate advisor, Prof. Jaejin Lee, for introducing me to research and helping come to graduate school. His thoughtful and valuable advices in various respects were the foundation for my graduate research.

I would like to show my gratitude to Prof. Susan Horwitz, Prof. Somesh Jha, Prof. Ben Liblit, and Prof. Karu Sankaralingam for serving on my final defense committee. Their encouragement, and insightful comments and questions helped me improve this thesis.

I have received so much help from past colleagues in Prof. Reps's research group. I thank Gogul Balakrishnan, Akash Lal, Nick Kidd, Denis Gopan, and Alexey Loginov for their constant advice and feedback that helped in developing my research work. Furthermore, I would like to thank present and former colleagues in PL group, Evan Driscoll, Aditya Thakur, Matt Elder, Tushar Sharma, Prathmesh Prabhu, Tycho Andersen, Emma Turetsky, Bill Harris, Ben Farley, Anne Mulhern, Cindy Rubio Gonzalez, Piramanayagam Arumuga, and Tristan Ravitch for always finding the time to attend my practice talks and give feedback. Also I would like to thank my present and former co-workers and advisors in GrammaTech, Prof. Tim Teitelbaum, David Melski, Suan Hsi Yong, Thomas Johnson, and Radu Gruian. Special thanks go to Prof. Shan Lu and Wei Zhang, with whom I enjoyed collaborating. Lastly, I offer my regards and blessings to all other friends who supported me in any respect during my graduate study.

My dissertation research was supported by NSF under grants CCF-0524051, CCF-0540955, CCF-0810053, and CCF-0904371; ONR under grants N00014-01-1-0708, N00014-01-1-0796, and N00014-09-1-0776; AFRL under contracts FA8750-06-C-0249 and FA8750-05-C-0179; a donation by GrammaTech, Inc.; and a Symantec Research Labs Graduate Fellowship.

Any opinions, findings, and conclusions or recommendations expressed in this document are those of the author, and do not necessarily reflect the views of the agencies and institutions that provided support for the work.

DISCARD THIS PAGE

TABLE OF CONTENTS

Page

Appendix

Page

DISCARD THIS PAGE

LIST OF TABLES

DISCARD THIS PAGE

LIST OF FIGURES

Appendix
Figure Page

Appendix
Figure Page

Appendix
Figure Page

Appendix
Figure Page

Appendix
Figure Page

ABSTRACT

As computers have become a pivotal component of daily lives, computer safety, reliability, and security issues have become enormously important. A considerable amount of recent research in program analysis and software engineering has been carried out on techniques and tools for finding software bugs and security vulnerabilities, and on checking computer-safety properties. Most of this research has focused on analyzing source code. Recently, machine-code analysis has begun to receive great attention both because source code is often unavailable and because there can be mismatches in various ways between source code and the machine code generated from the source code.

The tools and techniques for analyzing machine code are, in principle, language-independent. However, their implementations are often tied to one specific instruction set. Retargeting them to another instruction set can be an expensive and error-prone process. This dissertation describes a system that I developed, called TSL (for "**T**ransformer **S**pecification **L**anguage") that provides a systematic solution to the problem of creating retargetable tools for analyzing machine-code. The TSL system is a meta-tool, or tool generator, that automatically creates different abstract interpreters for machine-code instruction sets. The system addresses the problem of supporting multiple instruction sets by providing a YACC-like mechanism for creating key components of machine-code analyzers. The TSL system takes a single, unified description of the concrete operational semantics of an instruction set, which is specified in TSL, a strongly typed, first-order

functional language, and automatically creates implementations of different abstract interpreters for the given instruction set.

TSL provides a fixed set of base-types and operators, as well as map-types with map-access and (applicative) map-update operations. The TSL compiler generates a common intermediate representation that allows the meanings of the input-language constructs to be redefined by supplying alternative interpretations of the base-types, map-types, and the operations on them ("semantic reinterpretation"). Because all the abstract operations are defined at the *meta-level*, a semantic reinterpretation is independent of any given instruction set defined in TSL. Therefore, each implementation of an analysis component's driver serves as the unchanging driver for use in different instantiations of the analysis component for different instruction sets. The TSL language becomes the specification language for retargeting that analysis component to different instruction sets.

As an application of the TSL system, we developed a novel way of applying semantic reinterpretation to automatically create symbolic-analysis primitives for symbolic evaluation, weakest-liberal precondition, and symbolic composition. Furthermore, using the TSL system, as well as the TSL-generated symbolic-analysis primitives, we developed a machine-code verification tool, called MCVETO, and a concolic-execution-based program-exploration tool, called BCE.

- MCVETO addresses a large number of issues that arise when developing model-checking tools for machine code, for which standard techniques used in source-code model-checking tools would be unsound if applied to machine code.
- What distinguishes the work on BCE is that it makes use of control-dependence information to make program exploration goal-directed toward a given set of targets.

Chapter 1

Introduction

As computers have become a pivotal component of daily lives, computer safety, reliability, and security issues have become enormously important. To address these issues, a considerable amount of research has been carried out recently in the programming-language and software-engineering communities on techniques for finding software bugs and security vulnerabilities, and on checking computer-safety properties. This work has led to a large number of program-analysis techniques and tools. Essentially all of the results described in the literature are, in principle, language-independent; however, their implementations are often tied to one specific language. Retargeting them to another language (as well as implementing a new analysis for the same language) can be an expensive and error-prone process. For machine-code analyses, having a language-dependent implementation is even worse than for source-code analyses because instruction sets usually contain several hundred kinds of instructions, and a given instruction set often has special features not found in other instruction sets.

This dissertation describes a system that I developed, called TSL (for "**T**ransformer **S**pecification **L**anguage"), which helps in the creation of tools for analyzing machine code. The TSL system is a meta-tool, or tool generator, that automatically creates different abstract interpreters for machine-code instruction sets. The system addresses the problem of supporting multiple instruction sets by providing a YACC-like mechanism for creating key components of machine-code analyzers. The TSL system takes a single, unified description of the concrete operational semantics of an instruction set, and automatically creates implementations of different abstract interpreters for the given instruction set.

An instruction set's concrete semantics is specified in TSL's input language, which is a strongly typed, first-order functional language with a datatype-definition mechanism for defining recursive datatype, plus deconstruction by means of pattern matching. Writing a TSL specification for an instruction set is similar to writing an interpreter in first-order ML.

TSL provides a fixed set of base-types and operators, as well as map-types with map-access and (applicative) map-update operations. From a TSL specification, the TSL compiler generates a common intermediate representation (CIR) that allows the meanings of the input-language constructs to be redefined by supplying alternative interpretations of the base-types, map-types, and the operations on them ("semantic reinterpretation"). Because all the abstract operations are defined at the *meta-level*, semantic reinterpretation is independent of any given instruction set defined in TSL. Therefore, each implementation of an analysis component's driver serves as the unchanging driver for use in different instantiations of the analysis component to different instruction sets. The TSL language becomes the specification language for retargeting that analysis component for different instruction sets. Thus, to create $M \times N$ analysis components, the TSL system only requires M specifications of the concrete semantics of an instruction set, and N analysis implementations, i.e., $M + N$ inputs to obtain $M \times N$ analysis-component implementations.

Applications. As one application of the TSL system, we developed a novel way of applying semantic reinterpretation to automatically create symbolic-analysis primitives for symbolic evaluation, weakest-liberal precondition, and symbolic composition (see Chapter 4). Furthermore, using the TSL system, as well as the TSL-generated symbolic-analysis primitives, we developed a machine-code verification tool, called MCVETO (§5.1), and a concolic-execution-based program-exploration tool, called BCE (§5.2).

- MCVETO addresses a large number of issues that arise when developing model-checking tools for machine code, for which standard techniques used in source-code model-checking tools would be unsound if applied to machine code. These include (i) the absence of pre-built control-flow graphs and call graphs; (ii) the absence of meta data, such as information about variables, types, and aliasing; (iii) the absence of a fixed association between addresses and

instructions (e.g., *instruction aliasing* and *self-modifying code*); and (iv) extensive use of arithmetic on addresses in machine code.

- What distinguishes the work on BCE is that it makes use of control-dependence information to make program exploration goal-directed toward a given set of targets.

The remainder of this chapter is organized as follows: §1.1 discusses the challenge of software defects. §1.2 discusses a few of the approaches in the program-analysis literature that address the problem of software defects. §1.3 focuses on machine-code analysis and the challenges in implementing machine-code analyses. §1.4 presents an overview of the TSL system. §1.5 provides a short description of many of the applications to which I have applied TSL. §1.6 presents the organization of the dissertation and the contribution of each chapter.

1.1 The Challenge of Software Defects

Computers are pervasive in modern life, and are pivotal components in a wide range of contexts, such as (to name just a few) financial systems, power systems, manufacturing systems, asset-management systems, health-care systems, and many critical systems (e.g., nuclear reactors, weapons systems, and aircraft collision-avoidance systems). Computer-safety, reliability, and security issues have become enormously important because *software defects* and *security vulnerabilities* in computer systems can have severe consequences.

Software defects (bugs) and violations of computer-safety properties can cause critical failures (e.g., computer-system crashes) or other serious failures in a computer system (e.g., malfunctions due to mis-computations), which can result in severe damage both in financial terms and even in lives lost. We will mention just two cases of software bugs that had extreme consequences. For instance, in 1996, some problems with a rocket-launch software system caused a rocket that was set to deliver a payload of satellites into Earth orbit to veer off its path right after launch and self-destruct. This accident caused a loss of more than $370 million [3]. In one deadly incident in 1994 in Scotland, a system error caused a Chinook helicopter to crash, and all 29 passengers were killed [1].

Creating a correct and reliable computer system is becoming extremely difficult. As computer systems become more and more complex, even experienced software developers are prone to introducing bugs into their products; therefore, the potential for damage of the kind mentioned above continues to be a serious problem.

A security vulnerability in a software component is a flaw that can possibly be exploited by an adversary to create or install malware (such as viruses, worms, Trojans, bots, or back doors) or spyware; for conducting illegitimate activities, such as damaging and disrupting victims' computers; stealing confidential information, including passwords, credit-card numbers, and personal information; destroying important data; and even taking control of a compromised computer.

Annual worldwide economic damages from malicious software (malware) exceed $13 billion according to a survey conducted by Computer Economics in 2007 (available in the 2007 *Malware Report: The Economic Impact of Viruses, Spyware, Adware, Botnets, and Other Malicious Code* [2]). This amount includes only direct damages, such as loss of revenue due to loss or degraded performance of systems, labor costs to analyze, repair and cleanse infected systems, and loss of user productivity. Total damages would be substantially increased when indirect damages are considered.

Anti-malware technology is fairly effective in defending against many types of malware threats. However, traditional signature- and heuristic-based anti-malware technologies are often easily evaded, and thus no longer enough because there has been a significant increase in the number of *zero-day attacks* [22]. Zero-day attacks exploit security vulnerabilities that are unknown to others (including the original software developer), and for which no security fix is available at hand. Therefore, it becomes more important to detect and fix security vulnerabilities before software products are deployed, or before an adversary can exploit them to attack computer systems.

1.2 Program-Analysis Approaches

Program-analysis technology provides a promising approach to addressing the problems of finding bugs and security vulnerabilities, and for validating software systems. A considerable amount of recent research in the programming-languages and software-engineering communities

has led to techniques for (i) finding bugs, (ii) finding security vulnerabilities, and (iii) checking computer-safety properties. In these tools, program analysis conservatively answers the question "Can the program reach a bad state?". Although one cannot make an absolute distinction among those areas, which are closely related to each other, some of the related work in these areas can be summarized as follows:

- *Finding bugs/generating test cases.* DART (Directed Automated Random Testing) is a tool for automated testing [93]. To detect bugs that can cause program crashes and assertion violations, DART uses a combination of concrete execution and symbolic execution to systematically explore a program's state space. It uses symbolic execution to find inputs that direct execution along alternative paths.

  CBI (Cooperative Bug Isolation) is a feedback-directed approach to finding bugs [124]. Instrumented applications are deployed to the general public, and then some statistical methods are applied to mine returned data for information about root causes of failures.

  There are several other analysis tools for bug-finding and test-generation [52, 104, 119].
- *Finding security vulnerabilities.* BOON [181] is a static-analysis technique for determining whether a C program can index an array outside its bounds. CQual [88] is a type-based analysis tool that provides a lightweight, practical mechanism for specifying and checking properties of C programs. It uses type qualifiers to perform taint analysis, and detects format-string vulnerabilities in C programs. Eau Claire [64] is a tool for finding common security problems like buffer overflows, file-access race conditions, and format-string bugs. It uses a theorem prover to create a general specification-checking framework for C programs. Livshits proposed a static-analysis technique for detecting security vulnerabilities that stem from unchecked input in Java applications [130]. There are several other analysis tools for finding security vulnerabilities [37, 122].
- *Checking safety properties.* Havelund et al. presented a system called Java PathFinder, a model-checker for Java bytecode programs [100]. Ball et al. developed the Static Driver Verifier (SDV), which analyzes device-driver source code to determine whether there is a path in the driver that violates a kernel API usage rule [47]. MOPS uses model-checking

techniques to check certain kinds of security properties, represented as a finite-state automata [63]. There are many other analysis tools in this space [57, 63, 76, 102, 186]. These tools are based on *static analysis*, which is used to determine a conservative answer to the question "Can the program reach a bad state?"

These tools all focus on analyzing *source code* written in high-level languages, such as C, Java, etc. However, the problem of analyzing *machine code* to find bugs and security vulnerabilities, and to recover other information about their execution properties, has been receiving increased attention for the following reasons:

- Computers do not execute source code. Instead, the actual code that a computer executes is the machine code produced by a compiler (and an optimizer) from the source code. In the process of compiling and optimizing source code, subtle flaws depending on low-level, platform-specific details, such as memory layout, can be introduced. Consequently, there can be various vulnerabilities that are *invisible* in the original source code. Also, programs may be modified to insert malicious code. Balakrishnan et al. referred to such a situation as the WYSINWYX phenomenon (**W**hat **Y**ou **S**ee **I**s **N**ot **W**hat **Y**ou e**X**ecute) [38, 39, 41].
- Source code is often unavailable to analyze. For instance, Commercial-Off-The-Shelf (COTS) applications are typically delivered as stripped machine code (i.e., neither source code nor symbol-table/debugging information is provided). Also, malicious code such as bots and backdoors are in binary form and no source code for them is available.
- A program can be written in more than one language, which complicates the lives of developers of source-level tools. Also, when a program contains inlined assembly code, source-code analysis typically either ignores that part or does not push the analysis beyond it, which can make the results of the analysis unsound.
- Analyses based on source code typically make unchecked assumptions, e.g., that the program is ANSI-C compliant. This often means that an analysis does not account for behaviors that are allowed by the compiler (e.g., arithmetic is performed on pointers that are subsequently used for indirect function calls; pointers move off the ends of arrays and are subsequently dereferenced; etc.).

In these situations, the availability of good source-level analysis tools is irrelevant; instead, one needs tools capable of analyzing machine code.

1.3 Machine-Code Analysis

The aforementioned issues that arise when analyzing source code disappear when analyzing machine code. Furthermore, machine-code analysis has the advantage that it can provide more accurate information than a source-level analysis can because, for many programming languages, certain behaviors are left under-specified by the semantics. In such cases, a source-level analysis must account for all possible behaviors, whereas an analysis of machine code generally only has to deal with one possible behavior, namely, the one for the code sequence chosen by the compiler. Chapter 2 discusses machine-code analysis in more detail.

There have been several specialized analyses of machine code developed to identify aliasing relationships [80], data dependences [36, 70], targets of indirect calls [79], values of strings [68], bounds on stack height [159], and values of parameters and return values [190].

In contrast to such specialized analyses, Balakrishnan and Reps [38, 41] developed ways to address all of these problems by means of an analysis that discovers an over-approximation of the set of states that can be reached at each point in the executable—where a *state* means *all* of the components of a state: values of registers, flags, and the contents of memory. Moreover, their approach is able to be applied to stripped executables (i.e., neither source code nor symbol-table/debugging information need be available).

Challenges in implementing machine-code analysis. Machine-code analysis presents many new challenges. For instance, at the machine-code level, memory is one large byte-addressable array, and an analyzer must handle computed—and possibly non-aligned—addresses. It is crucial to track array accesses and updates accurately; however, the task is complicated by the fact that arithmetic and dereferencing operations are both pervasive and inextricably intermingled. For instance, if local variable `x` is at offset –12 from the activation record's frame pointer (register `ebp`), an access on `x` would be turned into an operand `[ebp–12]`. Evaluating the operand first involves

pointer arithmetic ("ebp–12") and then dereferencing the computed address ("[·]"). On the other hand, machine-code analysis also offers new opportunities, in particular, the opportunity to track low-level, platform-specific details, such as memory-layout effects. Programmers are typically unaware of such details; however, they are often the source of exploitable security vulnerabilities.

Many of the algorithms used in software model checkers that work on source code [47, 49, 102] would be unsound if applied to machine code. For instance, before starting the verification process proper, SLAM [47] and BLAST [102] perform flow-insensitive (and optionally field-sensitive) points-to analysis. However, such analyses often make unsound assumptions, such as assuming that the result of an arithmetic operation on a pointer always remains inside the pointer's original target. Such an approach assumes—without checking—that the program is ANSI C compliant, and hence causes the model checker to ignore behaviors that are allowed by some compilers (e.g., arithmetic is performed on pointers that are subsequently used for indirect function calls; pointers move off the ends of structs or arrays, and are subsequently dereferenced). A program can use such features for good reasons—e.g., as a way for a C program to simulate subclassing [172]—but they can also be a source of bugs and security vulnerabilities.

Although techniques developed in prior work on machine-code analysis are, in principle, language-independent, they have typically only been instantiated for one instruction set (mostly the Intel IA32 instruction set). This situation is actually typical of much work on source-code program analysis, too: even though the techniques described in the literature are, in principle, language-independent, their implementations are often tied to a specific language or intermediate representation. This state of affairs reduces the impact that good ideas developed in one context have in other contexts. The situation is more serious for low-level instruction sets, because (i) instruction sets usually contain several hundred instructions, and (ii) there are a variety of architecture-specific features that are incompatible with other architectures.

1.4 Transformer Specification Language (TSL)

To address the issues mentioned above, my work has aimed to provide a systematic way of implementing analyzers that work on machine code. As part of my research, I developed a language

for specifying the semantics of an instruction set, along with a run-time system to support dynamic analysis, static analysis, and symbolic analysis of executables written in that instruction set. This work advances the state of the art because it allows multiple analysis components to be created automatically from a single specification of the concrete operational semantics of the language to be analyzed. The system, called TSL (for "**T**ransformer **S**pecification **L**anguage"), has two classes of users: (1) instruction-set-specification (ISS) developers and (2) analysis developers. The former are involved in specifying the semantics of different instruction sets; the latter are involved in extending the analysis framework. In designing TSL, we were guided by the following principles:

- There should be a formal language for specifying the semantics of the language to be analyzed. Moreover, ISS developers should specify only the abstract syntax and a concrete operational semantics of the language to be analyzed—each analyzer should be generated automatically from this specification.
- Concrete syntactic issues—including (i) decoding (machine code to abstract syntax), (ii) encoding (abstract syntax to machine code), (iii) parsing assembly (assembly code to abstract syntax), and (iv) assembly pretty-printing (abstract syntax to assembly code)—should be handled separately from the abstract syntax and concrete semantics.[1]
- There should be a clean interface for analysis developers to specify the abstract semantics for each analysis. An abstract semantics consists of an *interpretation*: an abstract domain and a set of abstract operators (i.e., that performs abstract interpretations of the operations of TSL).
- The abstract semantics for each analysis should be separated from the languages to be analyzed so that one does not need to specify multiple versions of an abstract semantics for multiple languages.

Each of these objectives has been achieved in the TSL system: The TSL system translates the TSL specification of each instruction set to a common intermediate representation (CIR) that can be used to create multiple analyzers. Each analyzer is specified at the level of the meta-language (i.e.,

[1]The translation of the concrete syntaxes to and from abstract syntax is handled by a generator tool, called ISAL for Instruction Set Architecture Language, which is separate from TSL. ISAL was developed by GrammaTech [13].

by reinterpreting the operations of TSL), which—by extension to TSL expressions and functions—provides the desired reinterpretation of the instructions of an instruction set.

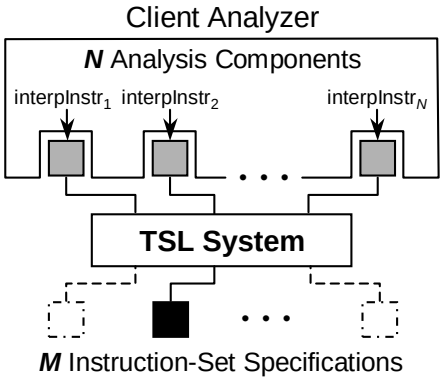


Figure 1.1 The interaction between the TSL system and a client analyzer. The grey boxes represent TSL-generated analysis components.

The TSL system provides two dimensions of parameterizability: different instruction sets and different analyses. Each ISS developer specifies an instruction-set semantics, and each analysis developer defines an abstract domain for a desired analysis by giving an interpretation (i.e., the implementations of TSL basetypes, basetype-operators, and map-access/update functions). Given the inputs from these two classes of users, the TSL system automatically generates an analysis component. Thus, to create $M \times N$ analysis components, the TSL system only requires M specifications of the concrete semantics of instruction sets, and N analysis implementations (Fig. 1.1), i.e., $M + N$ inputs are used to obtain $M \times N$ analysis-component implementations.

Many for the price of one! In Fig. 1.1, once one has the N analysis implementations that are the core of some client analyzer A, one obtains a generator that can create different versions A/M_1, A/M_2, ... at the cost of writing specifications of the concrete semantics of instruction sets M_1, M_2, etc. Thus, each client analyzer A created using analysis components generated via TSL acts as a "YACC-like" tool for generating different versions of A automatically.

$$s_1\colon x = x \oplus y;$$

$$s_2\colon y = x \oplus y;$$

$$s_3\colon x = x \oplus y;$$

Figure 1.2 Code fragment that swaps two `int`s;

1.4.1 Semantic Reinterpretation

The TSL system is based on factoring the concrete semantics of a language into two parts: (i) a *client* specification, and (ii) a semantic *core*. The interface to the core consists of certain base types, function types, and operators (sometimes called a *semantic algebra* [166]), and the client is expressed in terms of this interface. This organization permits the core to be *reinterpreted* to produce an alternative semantics for the *subject language*.[2]

Semantic Reinterpretation for Abstract Interpretation. The idea of exploiting such a factoring comes from the field of abstract interpretation [73], where factoring-plus-reinterpretation has been proposed as a convenient tool for formulating abstract interpretations and proving them to be sound [134, 144, 148]. In particular, soundness of the *entire* abstract semantics can be established via purely *local* soundness arguments for each of the reinterpreted operators.

The following example shows the basic principles of semantic reinterpretation in the context of abstract interpretation. We use a simple language of assignments, and define the concrete semantics and an abstract sign-analysis semantics via semantic reinterpretation.

Example 1.1 [Adapted from [134].] Consider the following fragment of a denotational semantics, which defines the meaning of assignment statements over variables that hold signed 32-bit `int`

[2] Semantic reinterpretation is a program-generation technique, and thus we follow the terminology of the partial-evaluation literature [108], where the program on which the partial evaluator operates is called the *subject program*.

In logic and linguistics, the programming language would be called the "object language". In the compiler literature, an object program is a machine-code program produced by a compiler, and so we avoid using the term "object programs" for the programs that TSL operates on.

values (where $\oplus$ denotes exclusive-or):

$$I \in \mathit{Id} \qquad E \in \mathit{Expr} ::= I \mid E_1 \oplus E_2 \mid \ldots$$
$$S \in \mathit{Stmt} ::= I = E; \qquad \sigma \in \mathit{State} = \mathit{Id} \to \mathit{Int32}$$
$$\mathcal{E} : \mathit{Expr} \to \mathit{State} \to \mathit{Int32}$$
$$\mathcal{E}[\![I]\!]\sigma = \sigma I$$
$$\mathcal{E}[\![E_1 \oplus E_2]\!]\sigma = \mathcal{E}[\![E_1]\!]\sigma \oplus \mathcal{E}[\![E_2]\!]\sigma$$
$$\mathcal{I} : \mathit{Stmt} \to \mathit{State} \to \mathit{State}$$
$$\mathcal{I}[\![I = E;]\!]\sigma = \sigma[I \mapsto \mathcal{E}[\![E]\!]\sigma]$$

By "$\sigma[I \mapsto v]$," we mean the function that acts like σ except that argument I is mapped to v. The specification given above can be factored into client and core specifications by introducing a domain *Val*, as well as operators *xor*, *lookup*, and *store*. The client specification is defined by

$$\mathit{xor} : \mathit{Val} \to \mathit{Val} \to \mathit{Val}$$
$$\mathit{lookup} : \mathit{State} \to \mathit{Id} \to \mathit{Val}$$
$$\mathit{store} : \mathit{State} \to \mathit{Id} \to \mathit{Val} \to \mathit{State}$$
$$\mathcal{E} : \mathit{Expr} \to \mathit{State} \to \mathit{Val}$$
$$\mathcal{E}[\![I]\!]\sigma = \mathit{lookup}\ \sigma\ I$$
$$\mathcal{E}[\![E_1 \oplus E_2]\!]\sigma = \mathcal{E}[\![E_1]\!]\sigma\ \mathit{xor}\ \mathcal{E}[\![E_2]\!]\sigma$$
$$\mathcal{I} : \mathit{Stmt} \to \mathit{State} \to \mathit{State}$$
$$\mathcal{I}[\![I = E;]\!]\sigma = \mathit{store}\ \sigma\ I\ \mathcal{E}[\![E]\!]\sigma$$

For the concrete (or "standard") semantics, the semantic core is defined by

$$v \in \mathit{Val}_{\mathit{std}} = \mathit{Int32} \qquad \mathit{lookup}_{\mathit{std}} = \lambda\sigma.\lambda I.\sigma I$$
$$\mathit{State}_{\mathit{std}} = \mathit{Id} \to \mathit{Val} \qquad \mathit{store}_{\mathit{std}} = \lambda\sigma.\lambda I.\lambda v.\sigma[I \mapsto v]$$
$$\mathit{xor}_{\mathit{std}} = \lambda v_1.\lambda v_2.v_1 \oplus v_2$$

Different abstract interpretations can be defined by using the same client semantics, but giving different interpretations to the base types, function types, and operators of the core. For example,

$$\sigma_0 := \{x \mapsto \mathit{neg}, y \mapsto \mathit{pos}\}$$
$$\sigma_1 := \mathcal{I}[\![s_1 : x = x \oplus y;]\!]\sigma_0 = \mathit{store}_{\mathit{abs}}\ \sigma_0\ x\ (\mathit{neg}\ \mathit{xor}_{\mathit{abs}}\ \mathit{pos}) = \{x \mapsto \mathit{neg}, y \mapsto \mathit{pos}\}$$
$$\sigma_2 := \mathcal{I}[\![s_2 : y = x \oplus y;]\!]\sigma_1 = \mathit{store}_{\mathit{abs}}\ \sigma_1\ y\ (\mathit{neg}\ \mathit{xor}_{\mathit{abs}}\ \mathit{pos}) = \{x \mapsto \mathit{neg}, y \mapsto \mathit{neg}\}$$
$$\sigma_3 := \mathcal{I}[\![s_3 : x = x \oplus y;]\!]\sigma_2 = \mathit{store}_{\mathit{abs}}\ \sigma_2\ x\ (\mathit{neg}\ \mathit{xor}_{\mathit{abs}}\ \mathit{neg}) = \{x \mapsto \top, y \mapsto \mathit{neg}\}.$$

Figure 1.3 Application of the abstract transformers created by the sign-analysis reinterpretation to the initial abstract state $\sigma_0 = \{x \mapsto \mathit{neg}, y \mapsto \mathit{pos}\}$.

for sign analysis, assuming that *Int32* values are represented in two's-complement notation, the semantic core is reinterpreted as follows:[3]

$$\begin{aligned}
v \in \mathit{Val}_{\mathit{abs}} &= \{\mathit{neg}, \mathit{zero}, \mathit{pos}\}^{\top} \\
\mathit{State}_{\mathit{abs}} &= \mathit{Id} \to \mathit{Val}_{\mathit{abs}} \\
\mathit{lookup}_{\mathit{abs}} &= \lambda\sigma.\lambda I.\sigma I \\
\mathit{store}_{\mathit{abs}} &= \lambda\sigma.\lambda I.\lambda v.\sigma[I \mapsto v]
\end{aligned}$$

$\mathit{xor}_{\mathit{abs}} = \lambda v_1.\lambda v_2.$

		v_2			
		neg	*zero*	*pos*	$\top$
v_1	*neg*	$\top$	*neg*	*neg*	$\top$
	zero	*neg*	*zero*	*pos*	$\top$
	pos	*neg*	*pos*	$\top$	$\top$
	$\top$	$\top$	$\top$	$\top$	$\top$

For the code fragment shown in Fig. 1.2, which swaps two `int`s, sign-analysis reinterpretation creates abstract transformers that, given the initial abstract state $\sigma_0 = \{x \mapsto \mathit{neg}, y \mapsto \mathit{pos}\}$, produce the abstract states shown in Fig. 1.3. □

Semantic Reinterpretation in TSL. The mapping of a client specification to the operations of the semantic core that one defines in a semantic reinterpretation resembles a translation to a

[3] For the two's-complement representation, $\mathit{pos}\ \mathit{xor}_{\mathit{abs}}\ \mathit{neg} = \mathit{neg}\ \mathit{xor}_{\mathit{abs}}\ \mathit{pos} = \mathit{neg}$ because, for all combinations of values represented by *pos* and *neg*, the high-order bit of the result is set, which means that the result is always negative. However, $\mathit{pos}\ \mathit{xor}_{\mathit{abs}}\ \mathit{pos} = \mathit{neg}\ \mathit{xor}_{\mathit{abs}}\ \mathit{neg} = \top$ because the concrete result could be either 0 or positive, and $\mathit{zero} \sqcup \mathit{pos} = \top$.

common intermediate representation (CIR) data structure. Thus, another approach to obtaining "systematic" reinterpretations that are similar to semantic reinterpretations—in that they apply to multiple subject languages—is to translate subject-language programs to a CIR, and then create various interpreters that implement different abstract interpretations of the node types of the CIR data structure. Each interpreter can be applied to (the translation of) programs in any subject language L for which one has defined an L-to-CIR translator. Compared with interpreting objects of a CIR data type, the advantages of semantic reinterpretation (i.e., reinterpreting the constructs of the *meta-language*) are

1. The presentation of our ideas is simpler because one does not have to introduce an additional language of trees for representing CIR objects.
2. With semantic reinterpretation, there is no explicit CIR data structure to be interpreted. In essence, semantic reinterpretation removes a level of interpretation, and hence generated analyzers should run faster.

1.4.2 Technical Contributions Incorporated in the TSL Compilation Process

The specific technical contributions incorporated in the part of the TSL compiler that generates the CIR can be summarized as follows:

- *Two-Level Semantics:* The notion of a *two-level* intermediate language [149] has been used to generate the CIR in a way that reduces the loss of precision that could otherwise come about with certain reinterpretation. To address this issue, the TSL compiler performs binding-time analysis [108] on the TSL specification to identify which values can always be treated as concrete values, and which operations should therefore be performed in the concrete domain (i.e., should not be reinterpreted). §3.2.1 discusses more details of the two-level intermediate language along with binding-time analysis.
- *Abstract Interpretation:* From a specification, the TSL compiler generates a CIR that has the ability (i) to execute over abstract states, (ii) possibly propagate abstract states to more than one successor in a conditional expression, (iii) compare abstract states and terminate abstract

execution when a fixed point is reached, and (iv) apply widening operators, if necessary, to ensure termination. §3.2.2 contains a detailed discussion of these issues.

- *Paired Semantics:* The TSL system allows easy instantiations of *reduced products* by means of *paired semantics*. The CIR can be instantiated with a *paired* semantic domain that couples two interpretations. Communication between the values carried by the two interpretations may take place in the TSL base-type operators. §3.2.3 discusses more details of paired semantics.

1.5 Overview of Applications of the TSL System

The capabilities of the TSL system have been demonstrated by writing specifications for both the IA32 and PowerPC instruction sets, and then automatically creating a variety of analysis components from each of the specifications—including dynamic-analysis components, static-analysis components and symbolic-analysis components from each of the specifications. The TSL-generated static-analysis components have been used to develop a parameterized version of CodeSurfer/x86. That is, using TSL, one can create CodeSurfer/M by writing a specification of the concrete semantics of instruction set M (§1.5.1). The dynamic-analysis and symbolic-analysis components generated using TSL have been used to develop the semantic primitives (§1.5.2) used in (parameterized versions of) a model-checking tool for machine code (§1.5.3) and a concolic-execution-based tool for analyzing bot executables (§1.5.4).

1.5.1 Static-Analysis Components

The TSL system has been applied to creating the analysis components employed by CodeSurfer/x86 [39], which is a static-analysis framework for analyzing stripped x86 executables. The TSL-generated analysis components include value-set analysis [38, 41], affine-relation analysis [38], def-use analysis (for memory, registers, and flags), and aggregate structure identification [42].

- **Value-Set Analysis** (VSA). VSA is a combined numeric-analysis and pointer-analysis algorithm that determines a safe approximation of the set of numeric values and addresses that

each register and memory location holds at each program point [41]. A *memory region* is an abstract quantity that represents all runtime activation records of a procedure. To represent a set of numeric values and addresses, VSA uses *value-set*s, where a value-set associates each memory-region with a map from abstract locations to strided intervals. A strided interval represents a set of numbers with a lower bound, an upper bound, and a stride [160].

- **Affine-Relation Analysis** (ARA). An affine relation is a linear-equality constraint between integer-valued variables. ARA finds all affine relationships that hold in the program, for a given set of variables. This analysis is used to find induction-variable relationships between registers and memory locations; these help in increasing the precision of VSA when interpreting conditional branches [38].
- **Aggregate-Structure Identification** (ASI). ASI is a unification-based, flow-insensitive algorithm to identify the structure of aggregates in a program [42]. For each instruction, the TSL-generated analysis component generates a set of ASI commands, each of which is either a command to *split* a memory region or a command to *unify* some portions of memory (and/or some registers). At analysis time, a client analyzer typically applies the generated ASI-command generator to each of the instructions in the program, and then feeds the resulting set of ASI commands to an ASI solver to refine the memory regions.
- **Quantifier-Free Bit-Vector** (QFBV) semantics. QFBV semantics provides a way to obtain a symbolic representation—as a formula in first-order quantifier-free bit-vector logic—of an instruction's semantics.
- **Def-Use Analysis** (DUA). Def-Use analysis collects all the *definitions* and *uses* of state components (memory-locations, registers, and flags) for each instruction.

These analysis components have been put together to create a system that essentially duplicates CodeSurfer/x86.

1.5.2 Symbolic-Analysis Components

Symbolic analysis has been an effective technique for testing and verifying programs because of the power that they provide in exploring a program's state space. The TSL system has been

applied to creating implementations of the basic primitives used in certain kinds of verification and testing tools that are based on symbolic program analysis. By "symbolic program analysis", we mean logic-based techniques to analyze state changes along individual program paths. This is in contrast to the situation addressed by many abstract-interpretation/dataflow-analysis techniques, which usually consider the problem of analyzing the effects of a *collection* of program paths—e.g., to identify program invariants. The basic primitives used in symbolic analysis are functions that perform *forward symbolic evaluation*, *weakest precondition*, and *symbolic composition* by manipulating formulas.

The conventional approach to implementing systems that use symbolic analysis is to write each of the three symbolic-analysis functions by hand for the programming language of interest. Our goal was to develop a method to create implementations of symbolic-analysis primitives easily, so that they can be made available for different subject languages—particularly for different machine-code instruction sets. Such instruction sets typically have (i) several hundred instructions, (ii) a variety of architecture-specific features that are incompatible with other architectures, and (iii) the ability to perform address arithmetic and dereferencing of addresses, which means that memory states can have complicated aliasing patterns. Consequently, our goal was to *generate* implementations of such primitives automatically from a specification of the subject language's concrete semantics.

Semantic reinterpretation for symbolic analysis. As a new application for semantic reinterpretation, we created implementations of the basic primitives used in symbolic program analysis. The aforementioned techniques and tools in the literature apply symbolic analysis to programs written in languages with pointers, aliasing, dereferencing, and address arithmetic. We demonstrate that the reinterpretation technique provides a way to create symbolic-analysis primitives for such languages.

With TSL each reinterpretation is defined at the *meta-level*, by reinterpreting the collection of TSL base types, function types, and operators. When a reinterpretation is performed in this way, it is independent of any given subject language. Consequently, with our implementation, all three of

the symbolic-analysis primitives can be generated automatically for *every* instruction set for which one has a TSL specification.

1.5.3 MCVETO: A Refinement-Based Model Checker for Machine Code

We used TSL to develop a model checker for machine code, called MCVETO (**M**achine-**C**ode **VE**rification **TO**ol). MCVETO uses *directed proof generation* [98] to find either an input that causes a (bad) target state to be reached, or a proof that the bad state cannot be reached. (The third possibility is that MCVETO fails to terminate.) What distinguishes the work on MCVETO is that it addresses a large number of issues that have been ignored in previous work on software model checking, and would cause previous techniques to be unsound if applied to machine code.

In our implementation, we restricted ourselves to use only language-independent techniques. In particular, we used a technique for generating automatically some of the key primitives of MCVETO's analysis components from descriptions of an instruction set's syntax and semantics [125, 126]—i.e., (a) an emulator for running tests, (b) a primitive for performing symbolic execution, and (c) a primitive for the pre-image operator. In addition, we developed language-independent approaches to the issues discussed above. Consequently, our system acts as a YACC-like tool for creating versions of MCVETO for different instruction sets: given an instruction-set description, a version of MCVETO is generated automatically. We created two such instantiations of MCVETO from descriptions of the Intel x86 and PowerPC instruction sets.

MCVETO is described in full detail in [174, 175]. §5.1 describes my contributions to MCVETO.

1.5.4 BCE: Analyzing Bot Executables

An increasing number of computers have been compromised by attacks from across the world to become part of malicious botnets [25]. Botnets seriously undermine computer security and reliability by conducting illegitimate activities, such as performing large-scale distributed denial-of-service attacks; identity theft; sending spam, trojans, and phishing emails; distributing pirated media; and performing click fraud. Moreover, botnets can quickly grow by using worms to attack

vulnerable systems. During the time between an announcement of a vulnerability and a patch for the vulnerability, the potential for bot infiltration is particularly high.

The Internet security research community has made significant efforts to identify botnets, to collect data on their activities, and to develop techniques for detection, mitigation, and disruption. Some bots try to avoid detection by using slow-spreading infection techniques. Some use multiple levels of indirection to make it harder to understand the botnet's structure. There have been several techniques to detect bots by monitoring network traffic to obtain temporal/spatial behavior statistics. Network-based and behavior-based approaches have several drawbacks: the approaches are (i) costly (runtime overhead to monitor network traffic, space overhead for storing packet logs, etc.), (ii) easily evaded, and (iii) not able to recover the structure of a botnet. Some detection techniques rely on well-known bot communication signatures: a lot of bot code is reused, and thus the commands and authentication mechanisms are widely known. However, attackers can easily modify the command-and-control language used by their bots to raise the bar for detection and control.

Using the TSL system, we have developed a tool called BCE for extracting botnet-command information from bot executables. BCE aims to provide useful information from analysis of bot executables by automatically extracting proper inputs that trigger malicious behavior. Applications of the information recovered include observing and analyzing malicious behaviors, as well as identifying command sequences that can be used at either the network or host level to mitigate botnets.

A typical way to analyze the behavior of a bot is to run the executable and observe its actions. To carry this out, however, one needs proper inputs that trigger malicious behaviors. Some widely known commands are often used for this purpose. However, attackers can easily change their commands to evade such an approach. It is a hard problem to obtain such inputs by manually stepping through the executable. BCE automates the extraction of information about botnet commands, and the arguments to commands, by driving the bot executable toward places where system calls are invoked.

In §5.2, we present BCE in detail.

1.6 Contributions and Organization of the Dissertation

The specific technical contributions of our work, along with the organization of the dissertation, are summarized as follows:

In Chapter 2, starting off the discussion on the advantages of machine-code analysis over source-code analysis and the challenges of machine-code analysis, we introduce CodeSurfer/x86 followed by an overview of two applications to which I applied CodeSurfer/x86—FFE/x86 and ConSeq. Lastly, we discuss the motivation for the main contribution of the dissertation, namely the TSL system.

In Chapter 3, we present the TSL system in detail. TSL will be presented from two perspectives: (i) how to write a TSL specification (from the point of view of instruction-set-specification developers), and (ii) how to write domains for (re)interpreting the TSL base-types (from the point of view of analysis developers). We also summarize the applications to which TSL has been used, including various static-analysis components that duplicate the hand-written ones used in CodeSurfer/x86, and discuss the leverage that we obtained through TSL.

In Chapter 4, we discuss the techniques that we developed to automatically create three symbolic-analysis primitives, and describe how the TSL system was used for that purpose. In particular, we show how semantic reinterpretation can be applied to create analysis functions that compute formulas for forward symbolic evaluation, weakest precondition, and symbolic composition.

In Chapter 5, we present case studies, including MCVETO, a model-checking tool for machine code, which uses the symbolic-analysis primitives generated from the TSL system, and BCE, a concolic-execution-based application, which extracts information about botnet commands from bot executables.

We present our conclusions in Chapter 6.

Chapter 2

Machine-Code Analysis

Computers do not execute source code; they execute machine code generated from source code by the combined efforts of the compiler, the optimizer, and the linker. The compiler and the optimizer make certain choices when generating machine code, depending on the target platform; therefore, there can be mismatches in various ways between what is actually executed on the processor and what a programmer really intends in her source code. Balakrishnan et al. refer to such a phenomenon as *WYSINWYX* ("What You See Is Not What You eXecute") ([41], [45] and [40, §1]).

The following example (obtained from [38]) shows a security vulnerability introduced due to the *WYSINWYX* phenomenon:

```
memset(password, 0, len);
free(password);
```

The password in clear text is stored in a dynamically-allocated buffer. Because the password is sensitive information, to minimize the lifetime of the password, the programmer tries to zero-out the buffer by calling `memset` before returning it to the heap by calling `free`. However, the `memset` call might be eliminated by a compiler that performs useless-code elimination, based on the reasoning that the program never uses the value written by the call on that function. Unfortunately, if this happens, sensitive information would be exposed in the heap.

As the above example illustrates, various vulnerabilities can be introduced by the compiler and the optimizer due to the idiosyncrasies inherited from a myriad of platform-specific features and various artifacts of the compiler and optimizer. These include (i) memory-layout details (i.e.,

offsets of variables in the run-time stack activation records and padding between fields of a struct), (ii) register usage, (iii) execution order, (iv) optimizations, and (v) artifacts of compiler bugs. Many security exploits make use of such artifacts [105, 135], and thus the target program can be executed by an attacker so that it operates differently (maliciously) from what is really intended by the programmer.

Such security vulnerabilities can escape the notice of tools that work on intermediate representations (IRs) that are built directly from the source code, whereas they are visible to analysis tools that work on machine code. In addition, there are a number of reasons why analyses based on source code do not provide the right level of detail for checking certain kinds of properties, and machine-code analyses do. Moreover, many issues arise when analyzing source code disappear when analyzing machine code: although Balakrishnan et al. have argued at length with examples the benefits of analyzing machine code rather than source code in [41], [45] and [40, §1], we summarize them in the following list:

- Source-level tools are only applicable when source code is available, which limits their usefulness in security applications (e.g., to analyzing code downloaded from the web or commercial off-the-shelf (COTS) applications, whose source code is usually unavailable). In particular, source-level tools cannot be applied to analyzing viruses and worms. Most applications are distributed as executables that have no symbol-table/debugging information ("stripped executables"). Although symbol-table/debugging information can be used to adapt source-level analysis techniques to work on machine code (when source code is unavailable), most analysis techniques are severely hampered when symbol-table/debugging information is absent.
- Even if source code is available, as discussed earlier, a substantial amount of information is hidden from analyses that start from source code, which can cause bugs, security vulnerabilities, and malicious behavior to be invisible to such tools. Moreover, a source-code tool that strives to have greater fidelity to the program that is actually executed would have to duplicate all of the choices made by the compiler and optimizer; such an approach would be extremely complicated to carry out. As alternative approach would be to use a compiler infrastructure,

such as LLVM [26] or GCC [12], that supports multiple compilers/optimizers. Such an approach would allow a source-code analysis tool to analyze the effects caused by compiler artifacts, but only for code created via the compiler infrastructure on which the analyzer is based. To make an analyzer *comprehensive* by mimicking multiple compilers/optimizers would require following such an approach for each possible compiler infrastructure—some of which are proprietary (e.g., the Microsoft Visual Studio compiler `cl`). In contrast, analyzing machine code directly provides a comprehensive solution: each run of the analyzer would give an answer for the machine-code program to which it is applied, but such an analyzer can be applied to machine-code programs produced by *any* compiler infrastructure, not just a particular one.

- Programs are sometimes modified subsequent to compilation, e.g., to perform optimizations or insert instrumentation code [182] or [112, 176]. Such modifications are not visible to tools that analyze source code.
- Machine-code analysis has an advantage that behavioral models derived from machine code can be *more accurate* than models derived from source code (particularly because compilation, optimization, and link-time transformation can change how the code behaves). Also, certain choices that the compiler and optimizer make can eliminate some possible behaviors—hence there is sometimes the opportunity to obtain more precise answers from machine-code analysis than from source-code analysis.
- Analyses based on source code typically make (unchecked) assumptions, e.g., that the program is ANSI-C compliant. This often means that an analysis does not account for behaviors that are allowed by the compiler (e.g., arithmetic is performed on pointers that are subsequently used for indirect function calls; pointers move off the ends of arrays and are subsequently dereferenced; etc.).
- Programs typically make extensive use of libraries, including dynamically-linked libraries (DLLs), which may not be available in source-code form. Typically, analyses are performed using code stubs that model the effects of library calls. Because these are created by hand, they are error-prone, and thus the analysis can return incorrect results. Because library code

can be analyzed directly in machine-code analysis, it is not necessary to rely on potentially-unsound models of library functions.[1]

- The source code may have been written in more than one language. This complicates the life of designers of tools that analyze source code because multiple languages must be supported, each with its own quirks.
- Even if the source code is primarily written in one high-level language, it may contain inlined assembly code in selected places. Source-level tools typically either skip over inlined assembly code [36] or do not push the analysis beyond sites of inlined assembly code [4]. Even if the source code was written in more than one language, a tool that analyzes executables only needs to support one language. Instructions inserted because of inlined assembly directives in the source code are visible, and do not need to be treated any differently than other instructions.
- An additional class of examples for which analysis of an executable can provide more accurate information than a source-level analysis arises because, for many programming languages, certain behaviors are left unspecified by the semantics. In such cases, a source-level analysis must account for all possible behaviors, whereas an analysis of an executable generally only has to deal with one possible behavior—namely, the one for the code sequence chosen by the compiler. For instance, in C and C++ the order in which actual parameters are evaluated is not specified: actuals may be evaluated left-to-right, right-to-left, or in some other order; a compiler could even use different evaluation orders for different functions. Different evaluation orders can give rise to different behaviors when actual parameters are expressions that contain side effects. For a source-level analysis to be sound, at each call site it must take the union of the descriptors that result from analyzing each permutation of the actuals. In contrast, an analysis of an executable only needs to analyze the particular sequence of instructions that lead up to the call.

[1] Machine-code analysis gives *platform-specific* answers. Models can be beneficial in obtaining answers that apply to multiple platforms by providing an answer relevant to all library versions that conform to the model.

2.1 Challenges in Machine-Code Analysis

Even though the advantages of analyzing executables are appreciated and well-understood, because of the obstacles standing in the way of doing a good job of machine-code analysis, there are a dearth of tools that work on executables directly. Compared with source-code analysis, analysis of stripped executables presents many challenges and difficulties, including

- *absence of information about variables:* In stripped executables, no information is provided about the program's global and local variables.
- *a semantics based on a flat memory model:* With machine code, there is no notion of separate "protected" storage areas for the local variables of different procedure invocations, nor any notion of protected fields of an activation record. For instance, a procedure's return address is stored on the stack; an analyzer must prove that it is not corrupted, or discover what new values it could have.
- *absence of type information:* In particular, `int`-valued and address-valued quantities are indistinguishable at runtime.
- *arithmetic on addresses is used extensively:* Moreover, numeric and address-dereference operations are inextricably intertwined, even during simple operations. For instance, consider the load of a local variable `v`, located at offset `-12` in the current activation record, into register `eax`: `mov eax,[ebp-12]`.[2] This instruction involves a *numeric* operation (`ebp-12`) to calculate an address whose value is then *dereferenced* (`[ebp-12]`) to fetch the value of `v`, after which the value is placed in `eax`.

[2]For readers who need a brief introduction to the 32-bit Intel x86 instruction set (also called IA32), it has six 32-bit general-purpose registers (`eax`, `ebx`, `ecx`, `edx`, `esi`, and `edi`), plus two additional registers: `ebp`, the frame pointer, and `esp`, the stack pointer. By convention, register `eax` is used to pass back the return value from a function call. In Intel assembly syntax, the movement of data is from right to left (e.g., `mov eax,ecx` sets the value of `eax` to the value of `ecx`). Arithmetic and logical instructions are primarily two-address instructions (e.g., `add eax,ecx` performs `eax := eax + ecx`). An operand in square brackets denotes a dereference (e.g., if `v` is a local variable stored at offset -12 off the frame pointer, `mov [ebp-12],ecx` performs `v := ecx`). Branching is carried out according to the values of condition codes ("flags") set by an earlier instruction. For instance, to branch to `L1` when `eax` and `ebx` are equal, one performs `cmp eax,ebx`, which sets `ZF` (the zero flag) to 1 iff `eax` $-$ `ebx` $= 0$. At a subsequent jump instruction `jz L1`, control is transferred to `L1` if `ZF` $= 1$; otherwise, control falls through.

- *instruction aliasing:* Programs written in instruction sets with varying-length instructions, such as x86, can have "hidden" instructions starting at positions that are out-of-registration with the instruction boundaries of a given reading of an instruction stream [128].
- *self-modifying code:* With self-modifying code there is no fixed association between an address and the instruction at that address.

Standard approaches to source-code analysis assume that certain information is available—or at least obtainable by separate analysis phases with limited interactions between phases, e.g.,

- a control-flow graph (CFG), or interprocedural CFG (ICFG)
- a call graph
- a set of variables, split into disjoint sets of local and global variables
- a set of non-overlapping procedures
- type information
- points-to information or alias information

The availability of such information permits the use of techniques that can greatly aid the analysis task. For instance, when one can assume that (i) the program's variables can be split into (a) global variables and (b) local variables that are encapsulated in a conceptually protected environment, and (ii) a procedure's return address is never corrupted, analyzers often tabulate and reuse explicit summaries that characterize a procedure's behavior.

Source-code analysis tools often use separate phases of (i) points-to/alias analysis (analysis of addresses) and (ii) analysis of arithmetic operations. Because numeric and address-dereference operations are inextricably intertwined, as discussed above, only very imprecise information would result if a machine-code analyzer used the same organization of analysis phases. Source-code-analysis tools sometimes also use questionable techniques, such as interpreting operations in integer arithmetic, rather than bit-vector arithmetic. They also usually make assumptions about the semantics that are not true at the machine-code level—for instance, they usually assume that the area of memory beyond the top-of-stack is not part of the execution state at all (i.e., they adopt the fiction that such memory does not exist).

2.1.1 CodeSurfer/x86

Because the problem of analyzing executables to recover information about their execution properties has been receiving increased attention, several techniques for analyzing machine code have been developed. However, much of this work has focused on *specialized* analyses to identify aliasing relationships [80], data dependences [36, 70], targets of indirect calls [79], values of strings [68], bounds on stack height [159], and values of parameters and return values [190].

In contrast to such specialized analyses, Balakrishnan and Reps [38, 41] developed ways to address all of these problems by means of an analysis that discovers an over-approximation of the set of states that can be reached at each point in the executable—where a *state* means *all* of the states: values of registers, flags, and the contents of memory. Their techniques have been incorporated into *CodeSurfer/x86* [5].

They have primarily been concerned with the analysis of stripped executables (i.e., neither source code nor symbol-table/debugging information is available), both because it is the most challenging situation, and because it is what is needed in the common situation where one needs to install a device driver or commercial off-the-shelf application delivered as stripped machine code. If an individual or company wishes to vet such programs for bugs, security vulnerabilities, or malicious code (e.g., back doors, time bombs, or logic bombs), analysis tools for stripped executables are required.

Some of the main analyses incorporated into CodeSurfer/x86 can be summarized as follows:

VSA VSA (*Value-Set Analysis*) provides useful information about memory accesses in an executable. VSA is a combined numeric-analysis and pointer-analysis algorithm that determines a safe approximation of the set of numeric values or addresses that each register and abstract memory location (*a-loc*) holds at each program point. In particular, at each program point, VSA provides information about the contents of registers that appear in an indirect memory operand; this permits it to determine the addresses that are potentially accessed, which, in turn, permits it to determine the potential effects on the state of an instruction that contains an indirect memory operand.

A key feature of VSA is that it tracks integer-valued and address-valued quantities simultaneously. This is crucial for analyzing executables because numeric values and addresses are indistinguishable at runtime.

ASI ASI (*Aggregate Structure Identification*) [42] is a unification-based, flow-insensitive algorithm to identify the structure of aggregates in a program. Whenever a read or write to a part of a memory object is encountered, ASI records how the memory object is to be subdivided into smaller objects that are consistent with the memory access.

The remainder of this chapter presents two analyzers that I developed that made use of, and extended, CodeSurfer/x86. §2.2 describes FFE/x86, which is a static-analysis tool for extracting an over-approximation of a program's output data format from an executable. §2.3 describes ConSeq, which is a consequence-oriented, backward-analysis framework for detecting concurrency bugs. ConSeq uses backward slicing obtained from CodeSurfer/x86 to identify shared memory reads that might impact each potential error site. §2.2 and §2.3 describe work that extended CodeSurfer/x86. §2.4 discusses the drawbacks of that approach, and presents the research goals for the work on the TSL system.

2.2 File-Format Extractor (FFE/x86)

Reverse engineering helps one gain insight into a program's internal workings. It is often performed to retrieve the source code of a program (e.g., because the source code was lost), to analyze a program that may be malicious (such as a virus), to fix a bug, to improve the performance of a program, and so forth. This section describes a reverse-engineering tool that can help a human understand what a program produces as its output.

The technique presented in this section promotes the reuse of components of a tool chain. For example, when a software engineer wants to build a program that can process the files that a COTS software product generates, he can use our tool to obtain information about the format specification, which would be useful when creating a program that can act as a substitute consumer (or producer).

Not all reverse-engineering activities are legal. One of the legal uses of reverse engineering is to obtain functional specifications needed for interoperability [29];[3] hence, the activity that our tool carries out would generally be considered a legitimate one.

The technique presented here might also be useful in malware detection. For instance, when trying to identify live versions of the same malware, one would like to have a way to figure out the format of its network traffic. Our technique can provide help with this problem.

Furthermore, our technique can provide a summary of a program's behavior: it produces a structure that consists of a reduced number of entities (compared with the call graph for instance), which may make it easier to understand what the program is doing.

We first construct a hierarchical finite-state machine [16, 34, 35] (HFSM) that represents a preliminary format structure, as explained in §2.2.3.1. However, an HFSM can be difficult to understand, so to increase the understandability of the results, we experimented with the application of several transformations (including simplification and regularization) to create an over-approximation of the HFSM as an ordinary finite-state machine (FSM), which represents a further over-approximation of the output data format. This can be used to present the final results either as an FSM or as a regular expression.

The contributions of the work described in this section are:

- It provides a technique for extracting an over-approximation of a program's output data format, including
  - a way to extract a preliminary structure for the output data format (§2.2.3)
  - a way to elaborate the structure by annotating it with information about possible output values and sizes (§2.2.4)
  - a way to simplify the structure to provide greater understanding of the output data format (§2.2.5)

  This provides information that can lead to greater understanding of a program's behavior.

[3] When a COTS (Commercial Off-The-Shelf) tool uses a proprietary file format, interoperability can be inhibited: the tool can only be used in a tool chain with a consumer or producer of files that have that format.

- We report experimental results from applying FFE/x86 on three applications. Our experiments uncovered a possible bug in `png2ico` (see [127] for details).

Although we have concentrated on the problem of extracting output file formats from executables, the same approach could be applied to source code (where one could also take advantage of information about the program's variables and their declared types), as well as to extracting input file formats.

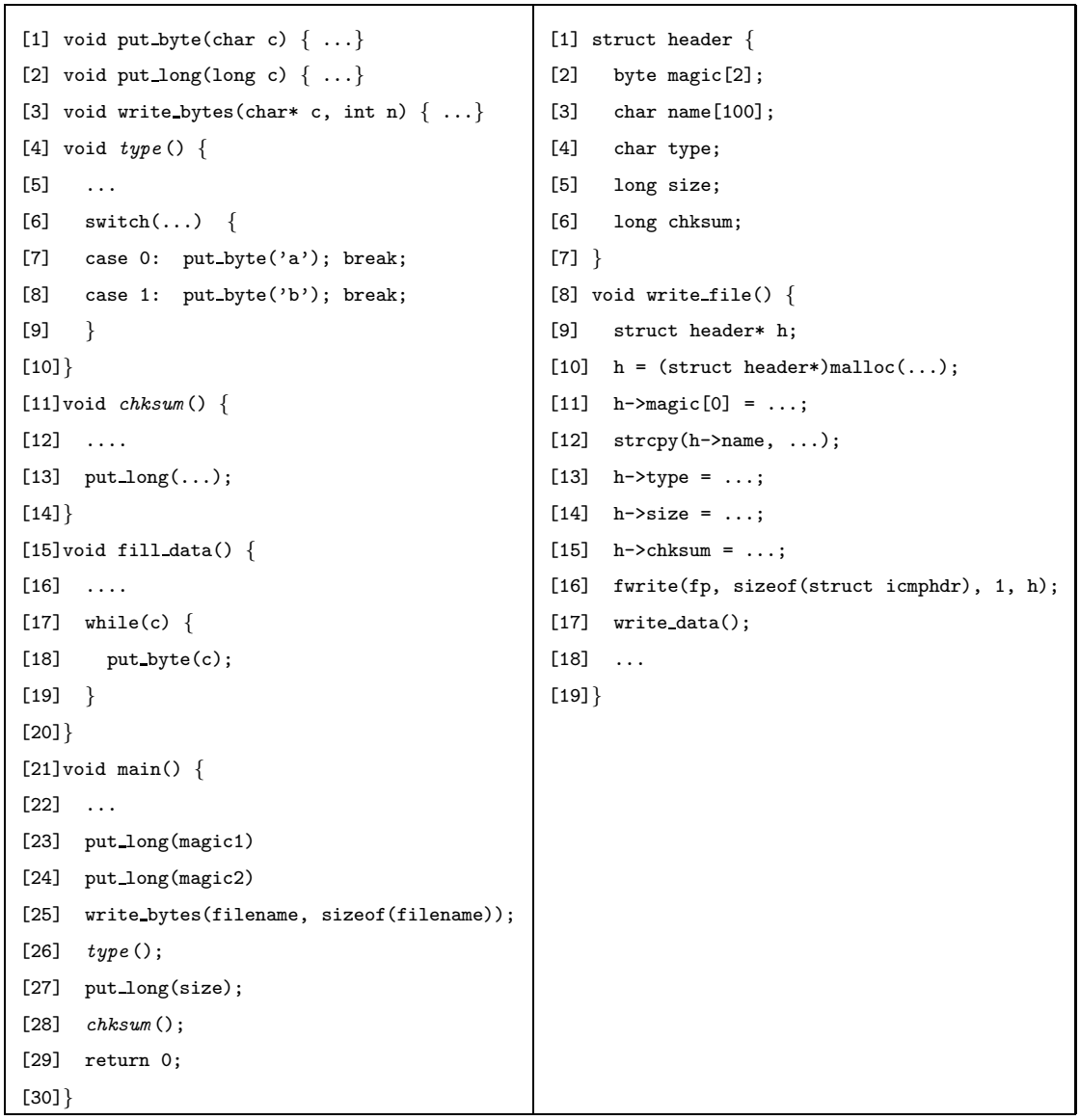
The remainder of this section is organized as follows: §2.2.1 discusses the key observations that inspired our work on FFE/x86 and the assumptions for our approach. §2.2.3 explains the process of constructing a structure for the output data format, and also provides an overview of the infrastructure on which our implementation is based. §2.2.4 discusses how to elaborate the structure generated from the first step with static analyses. §2.2.5 presents a series of filtering operations for making HFSMs more understandable. §2.2.6 describes how we validated FFE/x86. §2.2.7 presents experimental results. §2.2.8 describes related work. §2.2.9 describes possible future directions.

2.2.1 Programming Styles

This section makes a few observations about programming styles used in typical application programs to produce output data.

Programming styles relevant to writing output data can be categorized as *individual writes* and *bulk writes*. We present different approaches tailored to handle them in later sections. (Some programs use both styles; our tool is capable of handling such programs, as well.)

Individual writes. The first programming style is to write individual data items out separately to a file or a network. Standard I/O functions, such as *fputs* and *fputc* in C programs, could be used. In practice, however, *wrapper functions* tend to be frequently used. Fig. 2.1(a) shows an example of this programming style using wrapper functions, such as `put_byte`, `put_long`, and `writes`. Several fields of the output, including magic numbers, types, sizes, and a checksum, are written out by calling wrapper functions. These functions provide an API to append output items



```
[1] void put_byte(char c) { ...}
[2] void put_long(long c) { ...}
[3] void write_bytes(char* c, int n) { ...}
[4] void type() {
[5]   ...
[6]   switch(...)  {
[7]   case 0:  put_byte('a'); break;
[8]   case 1:  put_byte('b'); break;
[9]   }
[10]}
[11]void chksum() {
[12]  ....
[13]  put_long(...);
[14]}
[15]void fill_data() {
[16]  ....
[17]  while(c) {
[18]    put_byte(c);
[19]  }
[20]}
[21]void main() {
[22]  ...
[23]  put_long(magic1)
[24]  put_long(magic2)
[25]  write_bytes(filename, sizeof(filename));
[26]  type();
[27]  put_long(size);
[28]  chksum();
[29]  return 0;
[30]}
```

(a)

```
[1] struct header {
[2]   byte magic[2];
[3]   char name[100];
[4]   char type;
[5]   long size;
[6]   long chksum;
[7] }
[8] void write_file() {
[9]   struct header* h;
[10]  h = (struct header*)malloc(...);
[11]  h->magic[0] = ...;
[12]  strcpy(h->name, ...);
[13]  h->type = ...;
[14]  h->size = ...;
[15]  h->chksum = ...;
[16]  fwrite(fp, sizeof(struct icmphdr), 1, h);
[17]  write_data();
[18]  ...
[19]}
```

(b)

Figure 2.1 (a) An example that uses individual writes. (b) An example of a bulk write.

to an internal buffer; once the whole buffer has been filled, the contents of the buffer are flushed. Whereas the buffer is written out in bulk, the individual calls to the wrapper functions represent the "individual writes" referred to in our name for this style. We refer to both the standard I/O functions and user-defined wrapper functions as *output functions*.

An *output operation* is an operation relevant to generating an output data object. Specifically, the term output operation is defined as a call site that calls an output function—either a standard I/O library function or a wrapper function (see lines 7, 8, 13, 18, 23, 24, 25, and 27 in Fig. 2.1(a)).

Our experience so far is that many application programs are coded in this programming style. For instance, `gzip` [15],[4] `compress95` [6], and `png2ico` [20] follow such a programming style.

Bulk writes. The second programming style is to use `structs` or classes to manipulate headers. Fig. 2.1(b) shows an example of using a header structure to write output data. A header `struct` object is created at line 10. Each field of the `struct` is set to some value in lines 11–15. Finally, at lines 16–17, the object is written out to the file in its entirety. In this programming style, calls like the one to `fwrite` are the output operations.

In practice, we observed that `tar` [24] and `cpio` [8] use such aggregate structures as storage in preparation for a bulk write. We suspect that this style would be used for more than just headers by applications whose output files consist of a sequence of records.

2.2.2 User-Supplied Information

In our current implementation, the user must identify the output functions and supply some additional information about them, in particular, information about each output-relevant parameter:

- whether it is a numeric value to be written out
- whether it is an address pointing to the memory containing the data to be written out
- whether it indicates how many bytes are written out

See §2.2.4.1 for more details. In the case of standard I/O functions, such information is already known.

[4] Because the `gzip` source uses macros instead of functions, output operations are not call sites in the `gzip` executable. This is not compatible with our approach of having the user identify the output operations by supplying the names of output functions. To convert `gzip` into an example in which output operations are visible as procedure calls—so that it could be used for proof of concept in our experimental study—we modified the `gzip` source code to change all output macro definitions into explicit functions. Automatically identifying low-level code fragments that represent output operations remains a challenging problem for future work.

2.2.3 First step

In our approach, a *Hierarchical Finite State Machine (HFSM)* is used to represent an output data format. An HFSM is a structure in which nesting of finite automata within states is allowed [34, 35]. An HFSM captures commonalities by organizing states in such a hierarchy. Note the following two points about HFSMs:

- The languages of paths in recursive HFSMs are exactly the context-free languages.
- The languages of paths in non-recursive HFSMs are the regular languages.

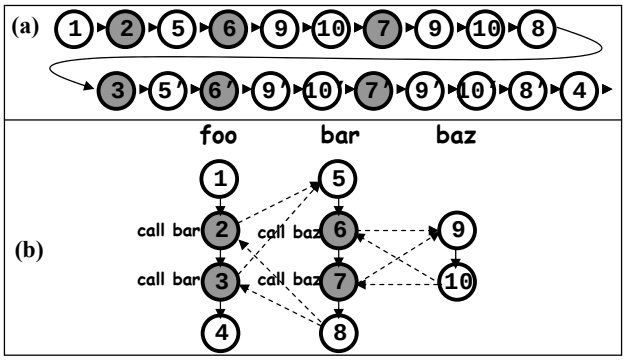


Figure 2.2 (a) An FSM, (b) A hierarchical FSM.

However, non-recursive hierarchical FSMs can be exponentially more succinct than conventional FSMs due to sharing, as illustrated in Fig. 2.2.

2.2.3.1 Construction of an HFSM

We will use the code fragment shown in Fig. 2.1(a) to explain our approach. The code emulates an archive utility. such as tar. It writes two magic numbers, followed by the file's name, layout type, size, and check-sum, using wrapper functions. Fig. 2.4 shows its disassembled code as generated by IDAPro [18], a commercial disassembly toolkit.

Each procedure involved with at least one output operation gives rise to an FSM. The program's wrapper functions include `put_byte` (`sub_401050` in the disassembled code), `put_long` (`sub_401075`), and `writes` (`sub_4010E4`), and calls to these functions represent output operations. FFE/x86 finds the output operations and constructs a hierarchical finite-state machine [16, 34, 35]

(HFSM) based on the control-flow graphs (CFGs) provided by the CodeSurfer/x86 framework mentioned in the introduction of this chapter [5]. Our analyzer creates a reduced interprocedural control-flow graph (i.e., the HFSM) that is the projection of the interprocedural control-flow graph onto enter nodes, exit nodes, call nodes, and output operations.

Fig. 2.3 shows the outcome from running FFE/x86. Each node in the HFSM is either an output operation (such as `4011B3`) or a call site (such as `4011D6`) to a sub-FSM (such as `type`). A call-site node, which represents a call to a sub-FSM, implicitly connects the two FSMs in the HFSM.

The HFSM generated by our tool for `gzip` is shown in Fig. 2.5(a). Our thesis is that HFSMs (including elaborations and refinements of HFSMs, as explained in §2.2.4 and §2.2.5) provide a basis for gaining an understanding of the program's behavior. In this regard, it is instructive to compare the HFSM with the program's call graph, because a call graph is another structure that a programmer may use to gain a high-level understanding of a program.

Fig. 2.5(b) shows a part of the call graph for `gzip`. `Gzip` is composed of 114 control-flow graphs (CFGs), 11491 CFG nodes, and 625 call sites. Even though the HFSM produced by our tool appears to be quite complicated, it is substantially less complicated than both the program's call graph and its interprocedural control-flow graph: the HFSM for `gzip` has 12 FSMs, 64 nodes, and 36 call sites.

2.2.3.2 Existing Infrastructure

FFE/x86 uses intermediate representations (IRs) provided by the CodeSurfer/x86 framework (Fig. 2.6), which provides an analyst with a powerful and flexible platform for investigating the properties and behaviors of x86 executables [5]. As described in the introduction of this chapter, CodeSurfer/x86 includes several static analyses, including VSA and ASI.

VSA is a combined numeric-analysis and pointer-analysis algorithm that determines an over-approximation of the set of numeric values and addresses that each memory location holds at each program point [41]. ASI recovers information about variables and types, especially for aggregates, including arrays and structs. The variables recovered by ASI are used by VSA to obtain information

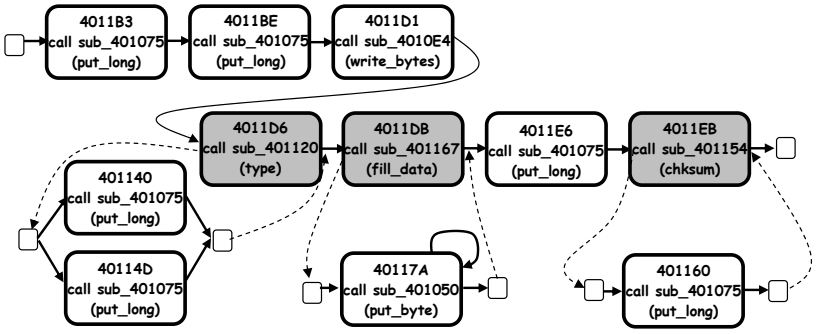


Figure 2.3 The HFSM for Fig. 2.1(a). The shaded boxes signify calls to FSMs. Dotted lines indicate implicit connections between FSMs.

```
401120 sub_401120 proc near;type
401120     push    ebp
401121     mov     ebp, esp
401123     sub     esp, 0Ch
401126     mov     eax, [ebp-4]
401129     mov     [ebp-8], eax
40112C     cmp     [ebp-8], 0
401130     jz      short loc_40113A
401132     cmp     [ebp-8], 1
401136     jz      short loc_401147
401138     jmp     short loc_401152
40113A loc_40113A:
40113A     mov     eax, [ebp-4]
40113D     mov     [esp], eax
401140     call    sub_401050
401145     jmp     short loc_401152
401147 loc_401147:
401147     mov     eax, [ebp-4]
40114A     mov     [esp], eax
40114D     call    sub_401050
401152 loc_401152:
401152     leave
401153     retn
401154 sub_401154 proc near;chksum
401154     push    ebp
401155     mov     ebp, esp
401157     sub     esp, 8
40115A     mov     eax, [ebp-4]
40115D     mov     [esp], eax
401160     call    sub_401075
401165     leave
401166     retn
401167 sub_401167 proc near;fill_data
401167     push    ebp
401168     mov     ebp, esp
40116A     sub     esp, 8
40116D loc_40116D:
40116D     cmp     [ebp-1], 0
401171     jz      short loc_401181
401173     movsx   eax, [ebp-1]
401177     mov     [esp], eax
40117A     call    sub_401050
40117F     jmp     short loc_40116D
401181 loc_401181:
401181     leave
401182     retn
401183 sub_401183 proc near;main
401183     push    ebp
401184     mov     ebp, esp
401186     sub     esp, 28h
401189     and     esp, 0FFFFFFF0h
40118C     mov     eax, 0
401191     add     eax, 0Fh
401194     add     eax, 0Fh
401197     shr     eax, 4
40119A     shl     eax, 4
40119D     mov     [ebp-14h], eax
4011A0     mov     eax, [ebp-14h]
4011A3     call    sub_401200
4011A8     call    __main
4011AD     mov     eax, [ebp-10h]
4011B0     mov     [esp], eax
4011B3     call    sub_401075
4011B8     mov     eax, [ebp-0Ch]
4011BB     mov     [esp], eax
4011BE     call    sub_401075
4011C3     mov     [esp+4], 4
4011CB     mov     eax, [ebp-8]
4011CE     mov     [esp], eax
4011D1     call    sub_4010E4
4011D6     call    sub_401120
4011DB     call    sub_401167
4011E0     mov     eax, [ebp-4]
4011E3     mov     [esp], eax
4011E6     call    sub_401075
4011EB     call    sub_401154
4011F0     mov     eax, 0
4011F5     leave
4011F6     retn
```

Figure 2.4 The disassembled code for Fig. 2.1(a). Transparent boxes indicate output operations, and shaded boxes indicate calls to sub-FSMs.

about the variables' possible values. The values recovered by VSA are used by ASI to identify a refined set of variables. Thus, CodeSurfer/x86 runs VSA and ASI repeatedly, either until quiescence, or until some user-supplied bound is reached.[5]

[5]If VSA and ASI have not quiesced when the bound is reached, it is still safe to use the results from the final round of VSA. In particular, each round of VSA provides an over-approximation of the set of numeric values and addresses

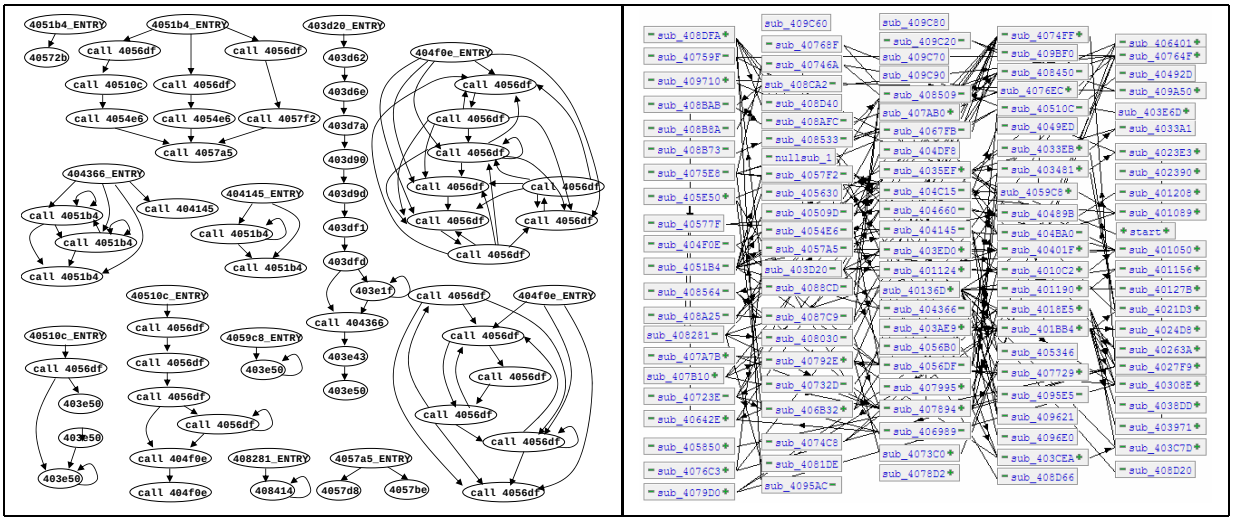


Figure 2.5 (a) The HFSM for `gzip`. (b) a fragment of the call graph of `gzip`.

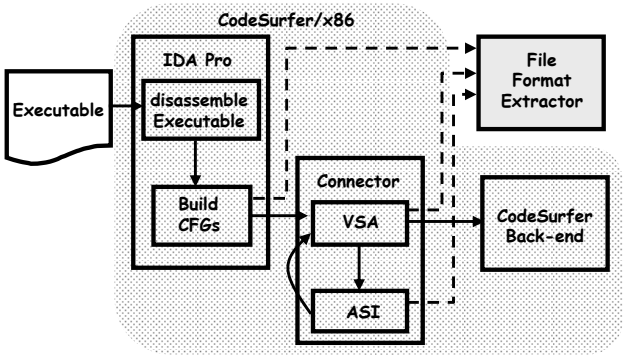


Figure 2.6 Organization of *CoderSurfer/x86*, and how FFE/x86 interacts with its components.

CodeSurfer/x86 uses an initial estimate of the program's variables, the call graph, and control-flow graphs (CFGs) for the program's procedures provided by IDAPro. IDAPro itself does not identify the targets of all indirect jumps and indirect calls, and therefore the call graph and control-flow graphs that it constructs are not complete. In contrast, CodeSurfer/x86 uses the values that VSA discovers to resolve indirect jumps and indirect calls, and thus is able to supply an over-approximation to the call graph.

for each memory location, modulo the treatment of possible memory-safety violations—some of which may be due to loss of precision during VSA. See [41] for more details.

§2.2.4 discusses other ways in which VSA and ASI can be exploited for our purposes.

2.2.4 Augmenting an HFSM with Information from Static Analyses

In this section, we explain how to exploit the static analyses mentioned in §2.2.3.2 for elaborating HFSMs.

2.2.4.1 Value Set Analysis

The HFSM generated by the method described in §2.2.3.1 provides some information for understanding an output format. The HFSM can be made more precise by annotating it with additional information. In particular, we wish to label each node with information about:

- the size (in bytes) of the data that the node represents, and
- an over-approximation of the value written out.

```
void put_byte(char c) {
 outbuf[outcnt++] = (uch)(c);
 if(outcnt==OUTBUFSIZE)
   flush_outbuf();
}
```

(a)

```
mov   byte ptr[esp], 1Fh
call  put_byte
```

(b)

Figure 2.7 An example code fragment; `put_byte` is a output function, and call sites that call it are output operations.

The values of interest are the actual parameters corresponding to the formal parameters of output functions. For example, suppose that `put_byte` is one of the output functions (see Fig. 2.7(a)). Suppose that at one of the call sites that calls `put_byte` (i.e., at one of the output operations), the actual parameter is always 1Fh (see Fig. 2.7(b)). This information can be obtained from the information collected by VSA. Note that at the call on `put_byte`, the relevant value is stored on the stack in the byte pointed to by `esp`. The abstract memory configuration (AMC) that VSA would have for the call site would indicate this: for instance, Fig. 2.8(a) illustrates the values that the AMC would contain in this example. In particular, our tool is able to obtain an over-approximation of the set

of values that the actual may hold by evaluating the operand expression `[esp]` in the AMC, which amounts to looking up in the AMC the contents of the cell (or cells) that `esp` may point to. (For this example, the result would be a singleton set, namely, {1Fh}.)

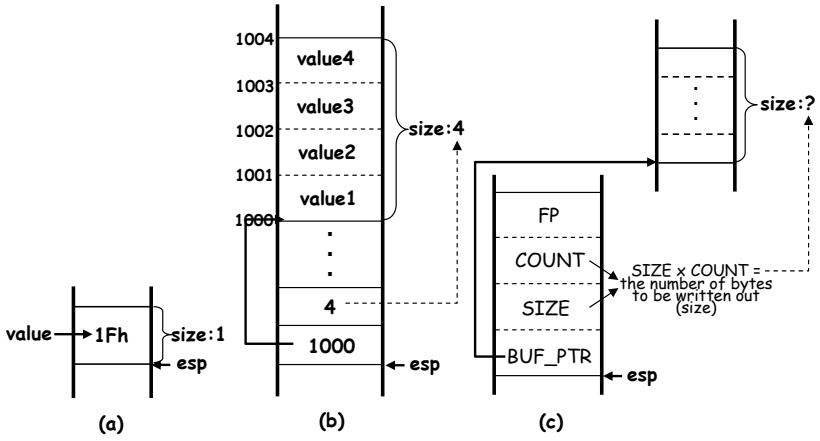


Figure 2.8 How to obtain information from VSA.

There are two kinds of parameters that can be passed into a output function: numeric values and addresses.

Numeric values. The case where an actual parameter holds a numeric value has already been explained above (see Fig. 2.8(a)). The corresponding size of the value can be obtained from ASI, which infers the size from the usage pattern of the formal parameter in the called function. (In the case where an output operation calls a standard I/O function, this information is available from the signature of the function.) For example, `put_byte` would have a 1-byte argument, `put_short` a 2-byte argument, and so forth.

Addresses. If the type of a formal parameter is a pointer, the set of addresses in the memory location corresponding to the actual parameter would be used to look up in the AMC the values in the cells to which the actual parameter could point (see Fig. 2.8(b)).

The case of `fwrite` at lines 16–17 in Fig. 2.1(b) falls into this category. The address of the heap-allocated memory location that contains the data is passed as the first argument.

`size_t fwrite(const void *`*BUF_PTR*`, size_t` *SIZE*`, size_t` *COUNT*`, FILE *`*FP*`);`

It is known that the product of the second and third parameters of `fwrite` is the number of bytes that are written out (see Fig. 2.8(c)).

Value roles. The kind of abstract value recovered by VSA sometimes suggests what the value's role is, e.g.,

- `Singleton` - If VSA recovers a singleton value for an actual parameter of an output operation, the parameter may correspond to either a magic number or a reserved field.
- `Set of numeric values` - If the value that VSA recovers is a non-singleton set of numeric values, the parameter may correspond to an optional field.
- `Top` - If VSA gives *Top*, which means any value, for an actual parameter of an output operation, the parameter may correspond to variant data.

2.2.4.2 Aggregate Structure Identification

As mentioned in §2.2.1, programmers frequently use a `struct` or a class to collect data before it is written out.

Fig. 2.9 shows a fragment from `ping` [19] in which a network packet is constructed. Instead of writing individual data items one at a time using output operations, a `struct` object is used to store output data while multiple fields are prepared, as shown in lines 7–11 of Fig. 2.9. Then the aggregate object is written out (i.e., sent out) all together on lines 13–14.

ASI [155] is a unification-based, flow-insensitive algorithm to identify the structure of aggregates in a program. Whenever a read or write to a part of a memory object is encountered, ASI records how the memory object should be subdivided into smaller objects that are consistent with the memory access.

In this example, we assume that the user has indicated that `sendto`, which is a GNU C library function, is the only output function. The second argument of `sendto` is known to be a pointer to a `struct` object with unknown substructure. ASI provides information about this substructure. The instructions that correspond to the assignment statements at lines 7–11 of Fig. 2.9 are shown in

```
[1] u_char outpack[MAXPACKET];
[2] static void pinger(void) {
[3]   register struct icmphdr *icp;
[4]   register int cc;
[5]   int i;
[6]   icp = (struct icmphdr*)outpack;
[7]   icp->icmp_type = ICMP_ECHO;
[8]   icp->icmp_code = 0;
[9]   icp->icmp_cksum = 0;
[10]  icp->icmp_seq = ntransmitted++;
[11]  icp->icmp_id = ident;
[12]  ...
[13]  i = sendto(s, (char*)outpack, cc, 0, &whereto,
[14]             sizeof(struct sockaddr));
[15]  ...
[16]}
```

Figure 2.9 Code fragment used to illustrate the use of ASI information.

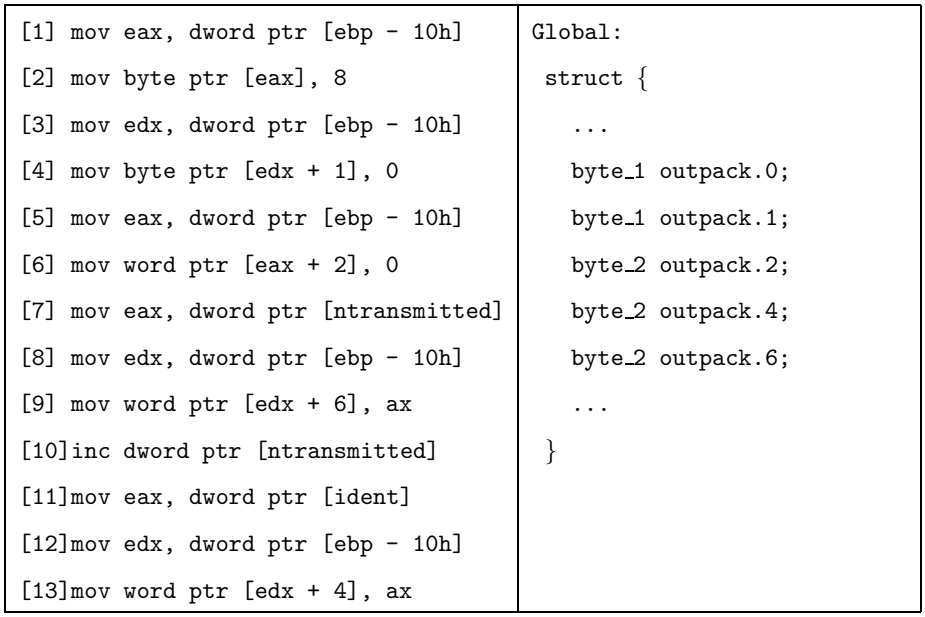
Fig. 2.10(a) at lines 2, 4, 6, 9, and 13, respectively. VSA provides information about the extent of memory accessed by each of these instructions. ASI uses that information to subdivide the portion of memory accessed, thereby producing the structure shown in Fig. 2.10(b). This indicates that the structure of the packet header may consist of two 1-byte fields, followed by three 2-byte fields.

ASI is also capable of recovering information about the structure of aggregates that are allocated in the heap.

This example illustrates a case where each output function emits a completely-constructed chunk of output data, and the HFSM represents the program's output operations at a high level of abstraction. In bulk writes as this example, structure information recovered by ASI can help identify the structure of output data format.

2.2.5 Filtering

Because an HFSM can be hard to understand, we experimented with applying a series of filtering operations—including simplification, conversion of each FSM to a regular expression, and



```
[1] mov eax, dword ptr [ebp - 10h]
[2] mov byte ptr [eax], 8
[3] mov edx, dword ptr [ebp - 10h]
[4] mov byte ptr [edx + 1], 0
[5] mov eax, dword ptr [ebp - 10h]
[6] mov word ptr [eax + 2], 0
[7] mov eax, dword ptr [ntransmitted]
[8] mov edx, dword ptr [ebp - 10h]
[9] mov word ptr [edx + 6], ax
[10]inc dword ptr [ntransmitted]
[11]mov eax, dword ptr [ident]
[12]mov edx, dword ptr [ebp - 10h]
[13]mov word ptr [edx + 4], ax
```

(a)

```
Global:
 struct {
   ...
   byte_1 outpack.0;
   byte_1 outpack.1;
   byte_2 outpack.2;
   byte_2 outpack.4;
   byte_2 outpack.6;
   ...
 }
```

(b)

Figure 2.10 (a) The disassembled code fragment for Fig. 2.9, (b) The outcome of ASI.

inline expansion—to generate a simpler representation of the output format as a regular expression. In our experiments, this has been done manually; however, the process would be relatively easy to automate.

Simplification. Not all nodes in the HFSM are helpful in understanding an output format. An unnecessarily complicated HFSM could prevent users from understanding key aspects of an output format.

Most portions of the HFSM shown in Fig. 2.5(a) turn out to be either `Top-value`, `Top-size`, or an unbounded loop that includes them. `Top-value` means that the node could have any value; `Top-size` means that the node could be of any size.

In each of the following cases, a node (or a node set) would not provide meaningful information:

- A node of `Top-size` and `Top-value`
- A node set in an unbounded loop, each of which has both `Top-size` and `Top-value`

To be considered as a *meaningful node*, a node must be

- A node of non-`Top-size`

Algorithm 1 Simplification algorithm.

Require: HFSM

Ensure: Trimed HFSM

Set the status of all FSMs to be *meaningful*

while There exists a *meaningful* FSM that contains only *non-meaningful nodes* or calls to *non-meaningful FSMs* **do**

Set *M* to be a *non-meaningful FSM*

Transform *M* into an FSM with a self-loop on a node labeled with (`Top-size/Top-value`)

end while

Alg. 1 describes an algorithm for simplifying HFSMs generated by FFE/x86. The idea behind the algorithm is to consider the cases mentioned above: for an FSM that consists of only nodes with `Top-value` and `Top-size`, or an unbounded loop that includes only such items, it may be better to simplify it to $(Top)^*$ because the original FSM would not provide much meaningful information about the output format.

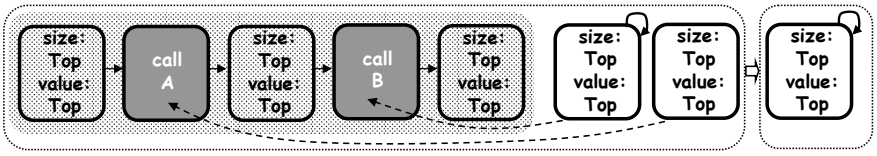


Figure 2.11 An example of simplification.

Fig. 2.11 shows an example of simplification. The shaded FSM that contains two *non-meaningful FSM*s and three *non-meaningful node*s is simplified to an unbounded self-loop consisting of a node (`Top-size`/`Top-value`).

Conversion to a regular expression. We can convert each FSM in an HFSM into a regular expression using the Kleene construction.

Expansion. The final step is to apply inline expansion. Recursion was not encountered in any of the applications that we used for our experiments (see §2.2.7), so inline expansion could be applied without worrying about non-termination. If recursion had been encountered, we could have summarized strongly connected components of the call graph.

Fig. 2.12 represents the final outcome from using these techniques on our example.

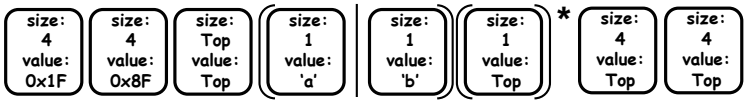


Figure 2.12 The final result after simplification, conversion, and inline expansion.

2.2.6 Validation against dynamic output

We validated our approach by testing whether the outcome from our algorithm (i.e., the regular expression) matches output data produced during actual runs of the application.

We used *flex* [11], a tool for generating scanners for compilers. Given an input specification in the form of a list of pattern-action pairs (where the pattern is a regular expression), *flex* generates a program that repeatedly finds the longest prefix of the (remaining) input that matches one of the patterns. To create a tool for testing whether a regular expression R generated by our algorithm describes the output of an application, we gave *flex* a 2-pattern specification—consisting of R (with an action to report success), plus a default pattern (with an action to report failure).

As discussed earlier, each box (as shown in Fig. 2.12) in the regular expression generated by our technique is labeled with two kinds of information: a value and a size. Value and size are either `Top`, a `Singleton`, or a set of numeric values. Thus, to be able to feed it to *flex*, the regular expression needs to be transformed to one in which the basic unit is a 1-byte character. Tab. 2.1 shows the transformation rules that are applied to boxes.[6]

[6]We use '.' as a shorthand for "any character". In *flex*, it is necessary to use the pattern '.|\n'.

Table 2.1 Transformation of boxes.

size	value	conversion
`Singleton` n	`Singleton`	According to the value of *n*, this is split into multiple boxes that contain a 1-byte value. (E.g., the first box in Fig. 2.13(a) is transformed to the first four boxes in Fig. 2.13(b).)
`Singleton` n	`Top`	`Top` is transformed to '.', which matches any character. Thus, this is transformed to a sequence of n boxes that contain '.'. (E.g., the fifth box in Fig. 2.13(a) is transformed to the last two boxes in Fig. 2.13(b).)
`Top`	`Top`	This is transformed to a box that contains '.' with a self-loop. (E.g., the third box in Fig. 2.13 (a) is transformed to the box that has a loop in Fig. 2.13(b).)

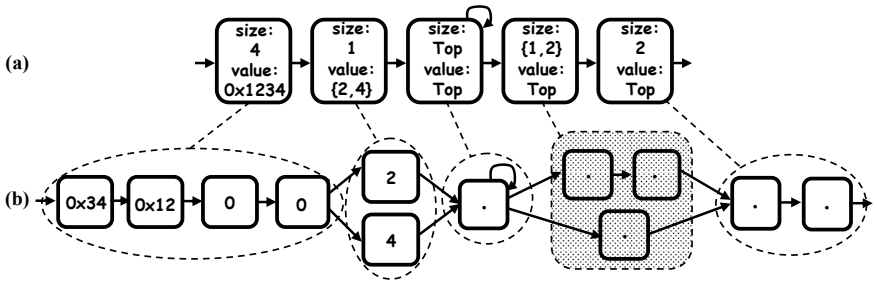


Figure 2.13 An example of the transformation. '.' means any character.

Tab. 2.1 describes only the cases when size and value have either `Singleton` or `Top`. (Note that there is no case when size is `Top` and the value is non-`Top` because this is not a possible outcome of VSA.) For the case when either size, value, or both have a set of numeric values, we split the box into multiple boxes that have a `Singleton` value and a `Singleton` size. For example, the second box in Fig. 2.13(a), which has two values (2 and 4), is transformed to the two boxes in Fig. 2.13(b) that have the values 2 and 4, respectively. For the case where size is not a `Singleton`, the shaded boxes in Fig. 2.13(b) show how it is converted.

Note that this process is only for validation, because the original values or sets of values are more likely to be understandable to a human than the subdivided values.

2.2.7 Experimental Results

We evaluated FFE/x86 on three applications: `gzip`, `png2ico`, and `ping`. In this chapter, we show the result of `gzip`. All the experimental resuls are presented in the WCRE'06 paper on FFE/x86 [127].

`Gzip`

`Gzip` is a GNU data-compression program. Fig. 2.14 represents the outcome after filtering the HFSM from Fig. 2.5(a).

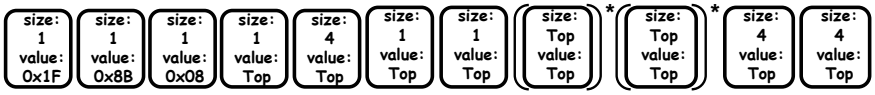
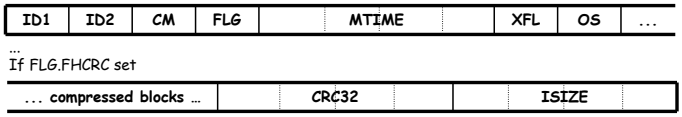


Figure 2.14 The final result for `gzip`.

Table 2.2 Part of the specification of `gzip`'s format [14].



ID1 and ID2	These are the fixed values: ID1=31 (0xlF), ID2=139 (0x8B)
CM	This identifies compression method: CM=0-7 are reserved, CM=8 demotes the "deflate" compression method.
FLG	This is divided into individual bits: bit 0 FTEXT, bit 1 FHCRC and so forth.
MTIME	This gives the most recent modification time of the original file being compressed.
XFL	This is available for use by specific compression methods.
OS	This identifies the type of file system on which compression took place: 0 - FAT filesystem, 1 - Amiga, and so forth.
CRC32	This contains a cyclic redundancy check value of the uncompressed data.
ISIZE	This contains the size of the original input data modulo 2^{32}.

The format of `.gz` files generated by `gzip` is described in RFC 1952 (see Tab. 2.2). The outcome shown in Fig. 2.14 correctly over-approximates the specification. In other words, the language of the outcome is a superset of the output language of `gzip`. The outcome has the two magic numbers (`ID1`=0x1f and `ID2`=0x8b) and a constant (`CM`=8) at the same positions shown in

Tab. 2.2. This is followed by a 4-byte element (corresponding to `MTIME`), two 1-byte elements (corresponding to `XFL` and `OS`). At the end, it has two 4-byte elements, which correspond to `CRC32` and `ISIZE`.

We also applied the validation process described in §2.2.6 to this outcome. The *flex*-generated validator accepted each of five `.gz` files (chosen arbitrarily from the Internet).

2.2.8 Related Work on Recovering Input/Output Information

Most previous work on reverse engineering of file formats has been dynamic and manual. Eilam describes a strategy for deciphering file formats given a symbol table and a sample output file [83]. This approach requires manually stepping through disassembled code and inspecting memory contents in a debugger while the program produces the given file. Other approaches ignore the program and rely on heuristic generalization from one or more sample output files. For example, one reverse-engineering case study searched for `zlib`-compressed data, file names, length bytes, and other typical structures [10]. All of these approaches require considerable manual effort and one cannot guarantee that the chosen sample files are sufficiently general. In constrast, the static approach described here over-approximates a file format without relying on sample files, symbol tables, or extensive manual analysis. Human intervention is only needed to identify output functions and to assign higher-level interpretations (e.g., “file name”) to selected fields identified by the analysis.

There have been similar attempts to statically recover information about program data. Christensen et al. have presented a technique for discovering the possible values of string expressions in Java programs [67]. First, a context-free grammar is generated by constructing dependence graphs from class files. The grammar is then widened into a regular language, which contains all possible strings that could be dynamically generated.

The method of Christensen et al. has also been applied to low-level code; Christodorescu et al. used the method in a string analysis for x86 executables [69]. This approach is similar to ours in the sense that x86 executables are the targets of both tools, and the recovered output data format in the analysis is represented as a regular language that denotes a superset of the actual output language.

Their approach, however, is different from ours in the sense that the initial context-free structure recovered by their tool comes from the structure of operations purely internal to each procedure, rather than from the call-return structure of the program, as in our tool.

Our approach is also related to work on host-based intrusion detection, in which models of expected program behavior are also constructed. The model over-approximates the possible sequences of system calls, and, by comparing the actual sequence of system calls to those allowed by the model, is used to detect when malicious input has hijacked the program. Pushdown-system models have been employed for this purpose, either constructed from source code [179] or from low-level code [91, 92] (in particular, SPARC executables). Our HFSMs are similar in that they also yield context-free languages that are a projection of a portion of the program's behavior. We have gone beyond previous work by using the results from two dataflow analyses (namely, VSA and ASI) to elaborate our models with information about possible sets of values and value sizes.

2.2.9 Discussion of FFE/x86

In the work on FFE/x86, we focus on output operations. However, the same approach can be applied to other kinds of operations. For example, one could treat *input operations*, which are associated with examining or parsing an input file, using the same approach taken by FFE/x86 [81]. In this case, one would want to consider only paths to exit points that represent successful runs of the program (because these correspond to successful uses of well-formed input files). In addition, one could apply our approach to network-communication operations that parse or construct packets.

It may be possible to use such a characterization of the input language as a way to generate test inputs. Similarly, knowledge of the output language for component c_1 in a tool chain could be used as a source of test inputs for the next component c_2 in the chain.

As mentioned earlier, we assume that output functions are identified by the user. To create a more automatic tool for extracting data formats, it would be desirable to find a way to automatically identify output functions, especially wrapper functions.

Each loop in an HFSM is currently transformed to either $(\texttt{node-set})^*$ or $(\texttt{node-set})^+$. However, there can be cases when the bound on the number of possible iterations of a loop can be

obtained from VSA. In such cases, the information about a loop's iteration bounds would provide users with more precise information about the output format.

More details can be found in the paper about FFE/x86 [127].

2.3 ConSeq

CodeSurfer/x86 has also been used as a component of a consequence-oriented backward-analysis framework, called ConSeq,[7] to detect *concurrency bugs* [191]. This section summarizes ConSeq, and describes the component for static slicing (§2.3.1), which was my contribution to the work.

Concurrency bugs are caused by non-deterministic interleavings between shared memory accesses. They exist widely (e.g., 20% of driver bugs examined in a previous study [162] are concurrency bugs) and are among the most difficult bugs to detect and diagnose because interleavings are not only complicated to reason about, but they also dramatically increase the state space of software. For large real-world applications, each input easily maps to billions of execution interleavings, and a concurrency bug may only be exposed by one specific interleaving. How to analyze this huge space *selectively* and expose hidden bugs is an open problem for static analysis, model checking, and software testing.

The effects of a bug propagate through data and control dependences until they cause software to crash, hang, produce incorrect output, etc. The lifecycle of a bug thus consists of three phases: (1) triggering, (2) propagation, and (3) failure. Traditional techniques for detecting concurrency bugs mostly focus on phase (1)—i.e., on finding certain structural patterns of interleavings that are common triggers of concurrency bugs. These patterns include data races (conflicting accesses to a shared variable) [66, 87, 147, 163, 189], simple atomicity violations (unserializable interleavings of two small code regions) [132, 151, 177, 188], context-switch bounded interleavings [56, 121, 142, 143], etc. Although much progress has been made in this direction, those techniques have

[7] ConSeq was carried out in collaboration primarily with W. Zhang, S. Lu, and T. Reps, along with R. Olichandran, J. Scherpelz, and G. Jin. My contribution to the work consisted of the development of the component for static slicing.

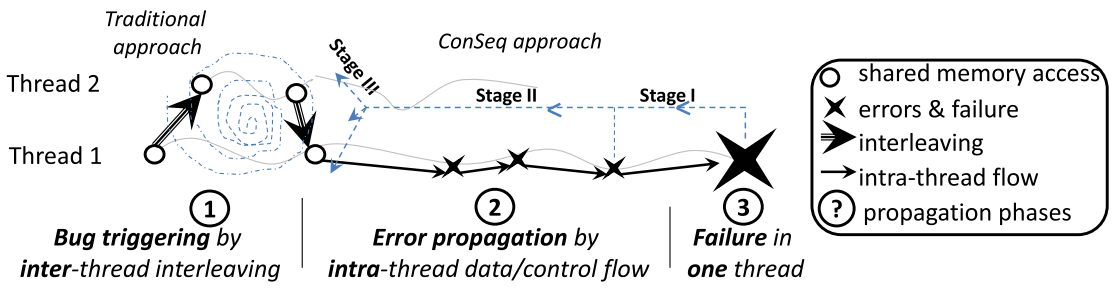


Figure 2.15 The common three-phase error-propagation process for most concurrency bugs (obtained from [191]).

fundamental limitations in that they can suffer from false negatives (i.e., many of common real-world concurrency bugs cannot be covered by traditional patterns) and false positives (i.e., the reported interleavings are not always truly harmful).

Consequence-Oriented Approach

To improve the accuracy and coverage of state-space search and bug detection, ConSeq is based on a consequence-oriented approach—that is, it uses a backwards approach, (3)→(2)→(1). ConSeq's backwards approach provides advantages in bug-detection coverage and accuracy but is challenging to carry out. ConSeq makes it feasible by exploiting the empirical observation that phases (2) and (3) usually are short and occur within one thread. ConSeq uses potential software failures to guide its search of the interleaving space. Our approach can be divided into the following three stages:

Stage I. ConSeq first statically identifies potential failure sites in an executable (i.e., it first considers a phase (3) issue). This approach is based on the observation that concurrency and sequential bugs have drastically different causes but have mostly *similar* consequences.

After being *triggered* by an incorrect execution order across multiple threads, a concurrency bug usually *propagates* in *one* thread through a short data/control-dependence chain, similar to one for a sequential bug [97]. The erroneous internal state is propagated until an externally visible failure occurs. At the end, concurrency and sequential bugs are almost indistinguishable: no matter

what the cause, a crash is often preceded by a thread touching an invalid memory location or violating an assertion; a hung thread is often caused by an infinite loop; incorrect outputs are emitted by one thread, etc.

ConSeq statically identifies **five** types of potential error sites that cover almost all major types of concurrency bug failures (**Stage I** of ConSeq as shown in Fig. 2.15): (1) calls to assertions in the software (for assertion crashes); (2) back-edges in loops (for infinite loop hangs); (3) calls to output functions (for incorrect functionality failures), (4) calls to error-message functions in the software (for various types of internal errors); and (5) reads on global variables where important invariants likely hold according to Daikon [85], a tool for inferring program invariants (for miscellaneous errors and failures).

Stage II. ConSeq then uses *static program slicing* from CodeSurfer/x86 to identify critical shared-memory read instructions that are highly likely to affect potential failure sites through a short chain of control and data dependences (phase (2)) (**Stage II** of ConSeq in Fig. 2.15).

ConSeq exploits two characteristics of concurrency bugs: first, the error-propagation distance is usually short in terms of data/control-dependence edges [97] (more information, including validation of the short-propagation heuristic can be found in [191]); second, the cause of a concurrency bug usually involves a specific ordering of just a few (two or three) shared memory accesses [56, 131].

§2.3.1 presents the details of Stage II.

Stage III. Finally, ConSeq monitors a single (correct) execution of a concurrent program, and by using execution-trace analysis and perturbation-based interleaving testing, it identifies suspicious interleavings that could cause an incorrect state to arise at a critical read and then lead to a software failure (phase (1)) (**Stage III** of ConSeq in Fig. 2.15).

ConSeq Modules

As shown in Fig. 2.16, ConSeq uses a combination of static and dynamic analyses. It uses the following modules to create an analyzer that works backwards along potential bug-propagation chains.

Error-site identifier. This static-analysis component processes an executable and identifies instructions where certain errors might occur. For example, a call to `__assert_fail` is a potential assertion-violation failure site. Although currently ConSeq identifies potential error sites for five types of errors, developers can adjust the bug-detection coverage and performance of ConSeq by specifying specific types of error sites on which to focus.

Critical-read identifier. This component uses static slicing to find out which instructions that read shared memory are likely to impact a potential error site. Note that static analysis is usually not scalable for multi-threaded C/C++ programs. By leveraging the short-propagation characteristic of concurrency bugs and the staged design of ConSeq, this module is scalable to large C/C++ programs. (§2.3.1 presents more details of this module.)

Suspicious-interleaving finder. This dynamic-analysis module monitors one run of the concurrent program, which is usually a correct run, and analyzes what alternative interleavings could cause a critical read to acquire a different and potentially dangerous value. By leveraging the characteristics of concurrency bugs' root causes, this module is effective for large applications. Via this module, ConSeq generates a bug report, which provides a list of critical reads that can potentially read dangerous writes and lead to software failures. Critical reads, dangerous writes, and the potential failure sites are represented by their respective program counters in the bug report. Additionally, the stack contents are provided to facilitate programmers' understanding of the bug report. [191] presents more details.

Suspicious-interleaving tester. This module tries out the detected suspicious interleavings by perturbing the program's re-execution. It helps expose concurrency bugs and thereby improves

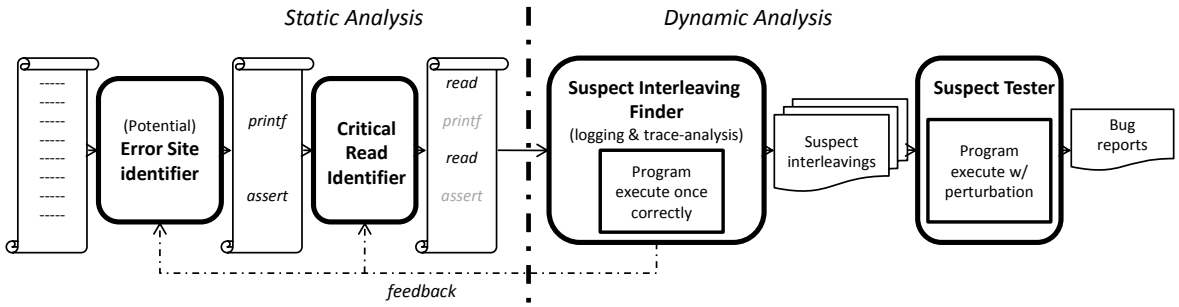


Figure 2.16 An overview of the ConSeq architecture (obtained from [191]).

programmers' confidence in their program. Via this module, ConSeq prunes false positives from the bug report, and extends the report of each true bug with how to perturb the execution and make the bug manifest. See [191] for more details.

2.3.1 Program Slicing in ConSeq

Program slicing is an operation that identifies semantically meaningful decompositions of programs, where the decompositions may consist of elements that are not textually contiguous [183]. A *backward slice* of a program with respect to a set of program elements S consists of all program elements that might affect (either directly or transitively) the values of the variables used at members of S. Slicing is typically carried out using *program dependence graphs* [103].

CodeSurfer/x86. ConSeq uses backward slicing to identify shared memory reads that might impact each potential error site. To obtain the backward slice for each potential error site, it uses CodeSurfer/x86 [39], which is a static-analysis framework for analyzing the properties of x86 executables. Various analysis techniques are incorporated in CodeSurfer/x86, including ones to recover a *sound approximation* to an executable's variables and dynamically allocated memory objects [41]. CodeSurfer/x86 tracks the flow of values through these objects, which allows it to provide information about control/data dependences transmitted via memory loads and stores.

The goal of the critical-read identification module is to identify *critical-read* instructions that are likely to impact potential error sites through data/control dependences. It uses static slicing to

approximate (in reverse) the second propagation phase of a concurrency bug, as shown in Fig. 2.15. The major design principle of this module is to only report instructions with short propagation distances as critical reads. Computing the complete program slice, e.g., all the way back to an input, is complicated and also unnecessary for ConSeq. ConSeq leverages the short-propagation characteristic of concurrency bugs to improve bug-detection efficiency and accuracy.

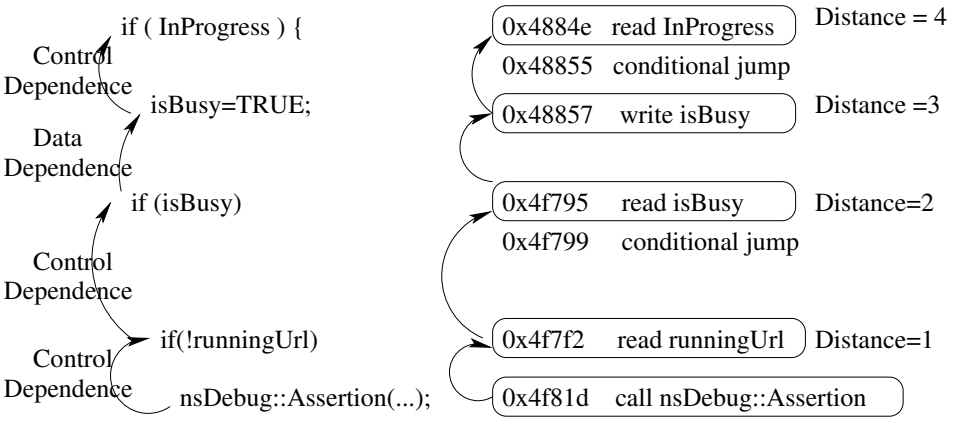


Figure 2.17 Static slicing (right) and the distance calculation (left; obtained from [191]).

In accordance with the short-propagation heuristic, ConSeq only reports read instructions whose return values can affect the error sites through a short sequence of data/control dependences. Our static-slicing tool provides the slice, together with the value of the shortest distance to the starting point of the slice, for each instruction of the slice. An example is shown in Fig. 2.17. ConSeq provides a tunable threshold *MaxDistance* for users to control the balance between false negatives and false positives. By default, ConSeq uses 4 as *MaxDistance*. A detailed evaluation is presented in [191].

Side-stepping scalability problems. To avoid the possible scalability problems that can occur with CodeSurfer/x86 due to the size of the applications used in evaluating ConSeq, we set the starting point of each analysis in CodeSurfer/x86 to the entry point of the function to which a given potential error site belongs, instead of the main entry point of the program. By doing so, CodeSurfer/x86 only needs to analyze the functions of interest and their transitive calls rather than the whole executable. Thus the static-analysis time grows roughly linearly in the number of

functions that contain error sites. This approach makes ConSeq much more scalable, as illustrated in the experimental section of [191].

This approach is applicable in ConSeq because—based on the observation that the error-propagation distance is usually short—ConSeq only requires a *short* backward slice that can be covered in one procedure. The backward-slicing and other analysis operations in CodeSurfer/x86 are, however, still context-sensitive and *interprocedural* [103]. Moreover, to obtain better precision from slices, each of the analyses used by CodeSurfer/x86 is also performed interprocedurally: calls to a sub-procedure are analyzed with the (abstract) arguments that arise at the call-site; calls are not treated as setting all the program elements to $\top$.

Analysis Accuracy. To obtain static-analysis results that over-approximate what can occur in any execution run, all the program elements (memory, registers, and flags) in the initial state with which each analysis starts are initialized to $\top$, which represents any value. Such an approximation makes sure that no critical read will be missed by ConSeq at runtime. Of course, some instructions could be mistakenly included in the backward slice and be wrongly treated as critical reads. Fortunately, our short-propagation-distance heuristic minimizes the negative impact of over-approximation. In practice, we seldom observe any inaccuracy caused by this over-approximation.

Identifying Potential Infinite Loop. For non-deadlock bugs, infinite loops in one thread are the main causes of hangs. Every back-edge in a loop is a potential site for this type of failure. ConSeq identifies strongly connected components (SCCs) that are potential failure sites for infinite-loop hangs by checking whether any shared-memory read is included in the backward slice of each back-edge in an SCC. To identify nested loops, CodeSurfer/x86 implements *Bourdoncle*'s algorithm [53], which recursively decomposes an SCC into sub-SCCs, etc.

More False-Positive Pruning via Symbolic Execution. The precision loss due to the properties of static analysis can result in spurious backward slices, which can cause false positives in ConSeq. To prune slices that are likely to be spurious, we introduce a heuristic based on *symbolic execution*,

which tracks symbolic expressions rather than actual values [62]. A symbolic execution is done by replaying a concrete trace produced from PIN [133], but executing it symbolically. Each trace must contain (i) a possibly false-positive critical read I and (ii) the control point B (conditional branch instructions) that controls execution of an error site. Two separate symbolic executions are performed for pruning: one *with* I (SE_1), and the other *without* I (SE_2). Each of the program elements is initialized to a symbol instead of a concrete value in the initial symbolic state with which each symbolic execution starts. We obtain the branching constraint C_1 from SE_1, and the second constraint C_2 from SE_2. If the following formula always holds, we can determine that I is a false positive (i.e., I does not impact the control toward the error site):

$$C_1 \Leftrightarrow C_2.$$

Due to the complexity of validity checking, we use the following formula as a heuristic:

$$S_1 \models C_2 \Leftrightarrow S_2 \models C_1$$

where S_1 and S_2 are satisfying assignments obtained using the YICES SMT solver for C_1 and C_2, respectively.[8]

2.3.2 Evaluation

The evaluation of ConSeq on large, real-world C/C++ applications shows that ConSeq detects more bugs than traditional approaches and has a much lower false-positive rate [191]. ConSeq was evaluated on 11 real-world concurrency bugs in seven widely used C/C++ open-source server and client applications—Mozilla, MySQL, Cherokee, Transmission, Aget, etc. ConSeq was able to detect 10 out of 11 tested concurrency bugs, which cover a wide range of root causes, from simple races and single-variable atomicity-violations to order-violations, anti-atomicity violation bugs, multi-variable synchronization problems, etc. For comparison, we evaluated a race detector and an atomicity-violation detector and found that they could only detect 3 and 4 bugs, respectively.

[8] For the implementation of this particular part, we use symbol-analysis primitives (symbolic-execution primitive and satisfaction relation) created by TSL [126, 125], which is the main subject of this thesis. TSL will be presented in the following chapters.

ConSeq detected these bugs with high accuracy: it had about one-tenth the false-positive rate of the race detector and the atomicity-violation detector.

ConSeq also found 2 new bugs in Aget, 2 new bugs in Click, and one output non-determinism in Cherokee, for which bugs had not been previously reported. ConSeq found a known infinite-loop bug in a version of MySQL for which the bug had not been previously reported. Experiments in which we used ConSeq together with Daikon [85] show that ConSeq can detect complicated concurrency bugs that previous tools cannot (e.g., a bug involving 11 threads and 21 shared variables). The performance of ConSeq is suitable for in-house testing.

More details of the experimental results are presented in the ASPLOS'11 paper about ConSeq [191].

2.3.3 Discussion of ConSeq

The work on ConSeq provides a new perspective on concurrency-bug detection and testing, which is to start from potential consequences and work backwards. It provides alternative interpretations for some concurrency bugs with complicated causes that are difficult to detect using traditional approaches, and sets up a nice connection with sequential bug-detection research, such as Daikon [85].

ConSeq uses a three-stage bug-detection framework that leverages characteristics from all three phases of the concurrency-bug propagation process. The design separates the complexity of inter-thread interleaving analysis and intra-thread propagation analysis, and makes it easy to leverage advanced static-analysis techniques, such as slicing and loop analysis. Each stage of the framework can be easily extended. In particular, programmers can assist ConSeq by putting more consistency checks into their code, such as assertions and error messages.

Overall, ConSeq effectively exposes those non-determinisms among a small number of shared memory accesses that can propagate a relatively short distance and cause a common error (such as infinite loop, error message firing, assertion failure, etc.) and end up with a visible failure.

2.4 Motivation for a New System for Implementing Machine-Code Analyses

Although the analysis techniques incorporated into CodeSurfer/x86, in principle, are language-independent, the original implementation was tied to the Intel IA32 instruction set. Moreover, CodeSurfer/x86 incorporated at least eight separate analyses, each of which was an independently-coded abstract interpretation of the IA32 instruction set's concrete semantics. Fig. 2.18 shows some simplified versions of the implementations of VSA (on the left) and ASI (on the right) in CodeSurfer/x86. The implementation of the abstract transformer for each analysis usually has a big switch statement where for each instruction of IA32, an abstract transformer is implemented in the analysis abstract domain according to the concrete semantics of the instruction. The switch statement for each analysis in CodeSufer/x86 contains about 110 cases[9] for frequently-used IA32 instructions. If one wanted to develop, e.g., CodeSurfer/PowerPC, substantial work would be necessary to port the original CodeSurfer/x86 implementation to support a new instruction set. In particular, CodeSurfer/x86 consists of eight analyses, and an abstract transformer for each instruction of PowerPC would need to be implemented for each of the eight analyses' abstract domains.

In general, if one can has N subject languages and a desired tool that consists of M analysis components, one would have to create $N \times M$ analysis-component implementations. (One of the advantages of the TSL system is that to obtain the desired $N \times M$ analysis-component implementations, a human tool designer will only have to perform $N + M$ work.)

The situation described above is fairly typical of much work on program analysis: although the techniques described in the literature are, in principle, language-independent, implementations are often tied to a specific language or intermediate representation (IR). Retargeting them to another language can be an expensive and error-prone process. Even for source-code analysis, this state of affairs reduces the impact that good ideas developed in one context (e.g., Java program analysis) have in other contexts (e.g., C++ analysis).

For high-level languages, the situation has been addressed by developing common intermediate languages, e.g., GCC's RTL, Microsoft's MSIL, etc. (although the academic research community

[9] The remaining instructions out of about 600 IA32 non-floating-point/non-MMX instructions are treated as causing the resultant state to be Top.

```
[1]  VSA_state_t VsaTransformerForIA32(
[2]      Instr i, VSA_state_t S)
[3]  {
[4]    VSA_state_t ans;
[5]    switch(i.id) {
[6]    case IA32_MOV: {
[7]      VSA_value_t v = EvalVSA(i.child2, S);
[8]      ans = UpdateVSAState(S, i.child1, v);
[9]      break;
[10]   }
[11]   case IA32_ADD: {
[12]     VSA_value_t v1 = EvalVSA(i.child1, S);
[13]     VSA_value_t v2 = EvalVSA(i.child2, S);
[14]     VSA_value_t v = VSAPlus(v1, v2);
[15]     ans = UpdateVSAState(S, i.child1, v);
[16]     break;
[17]   }
[18]   case IA32_SUB: {
[19]     ...
[20]     break;
[21]   }
[22]   }
[23]   return ans;
[24]}
```

```
[1]  set_of_mini_asi_instr AsiTransformerForIA32(
[2]      Instr i, VSA_state_t S)
[3]  {
[4]    set_of_mini_asi_instr ans;
[5]    switch(i.id) {
[6]    case IA32_MOV: {
[7]      set_of_mini_asi_instr v1 =
[8]          CollectMemAccesses(i.child1, S);
[9]      set_of_mini_asi_instr v2 =
[10]         CollectMemAccesses(i.child2, S);
[11]     ans = v1.union(v2);
[12]     break;
[13]   }
[14]   case IA32_ADD: {
[15]     ...
[16]     break;
[17]   }
[18]   case IA32_SUB: {
[19]     ...
[20]     break;
[21]   }
[22]   }
[23]   return ans;
[24]}
```

Figure 2.18 Two snippets of VSA and ASI implementations in CodeSurfer/x86; EvalVSA/ UpdateVSAState and CollectMemAccesses are other IA32-specific procedures for VSA and ASI, respectively; ASI makes use of the information from VSA.

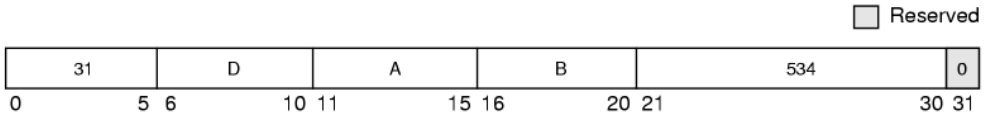
has not rallied around a similar common platform). The situation is more serious for low-level instruction sets, because (i) most instruction sets have evolved over time, so that each instruction-set

lwbrx **lwbrx**

Load Word Byte-Reverse Indexed

lwbrx **r**D,**r**A,**r**B

[POWER mnemonic: **lbrx**]




```
if rA = 0 then b ← 0
else      b ← (rA)
EA ← b + (rB)
rD ← (32)0 || MEM(EA + 3, 1) || MEM(EA + 2, 1) || MEM(EA + 1, 1) || MEM(EA, 1)
```

EA is the sum (**r**A|0) + **r**B. Bits 0–7 of the word in memory addressed by EA are loaded into the low-order 8 bits of **r**D. Bits 8–15 of the word in memory addressed by EA are loaded into the subsequent low-order 8 bits of **r**D. Bits 16–23 of the word in memory addressed by EA are loaded into the subsequent low-order eight bits of **r**D. Bits 24–31 of the word in memory addressed by EA are loaded into the subsequent low-order 8 bits of **r**D. The high-order 32 bits of **r**D are cleared.

The PowerPC architecture cautions programmers that some implementations of the architecture may run the **lwbrx** instructions with greater latency than other types of load instructions.

Other registers altered:

- None

Figure 2.19 The description of the PowerPC instruction `lwbrx` (obtained from the PowerPC instruction-set manual [27]).

family has a bewildering number of variants,[10] which has led to instruction sets with several hundred instructions, and (ii) there are a variety of architecture-specific features that are incompatible with other architectures.

Fig. 2.19 shows an informal description of the operational semantics of an instruction in the 32-bit PowerPC instruction set. One can imagine how expensive and error prone it would be to develop an analysis implementation because the developer needs to interpret the instruction's concrete semantics in the abstract domain used by the analysis.

[10]For a brief overview, see http://en.wikipedia.org/wiki/{X86,ARM_architecture,PowerPC}. In particular, the article about ARM lists 25 different architectural versions [Sept. 29, 2008].

Our motivation is to provide a systematic way of extending the analyses used in CodeSurfer/x86—and others—to instruction sets other than IA32. The motivation led us to develop a meta-tool (or tool-generator), called TSL (for "**T**ransformer **S**pecification **L**anguage"), to help in the creation of tools for analyzing machine code. TSL consists of a language for describing the semantics of an instruction set, along with a run-time system to support the static analysis of executables written in that instruction set. The work advances the state of the art by creating a system for automatically generating analysis components from a specification of the language to be analyzed. In the remaining chapters, we introduce TSL and describe some of its capabilities.

Chapter 3

Transformer Specification Language

In Chapter 2, we discussed the importance and advantages of machine-code analysis and challenges in developing a system for analyzing machine-code. This chapter presents the TSL system that we have developed to address the challenging issues discussed in §2.4. "TSL" stands for "**T**ransformer **S**pecification **L**anguage", and is used both for the name of the overall system and for the name of the system's meta-language.

Design Principles

In designing TSL, we were guided by the following principles:

- There should be a formal language for specifying the semantics of the language to be analyzed. Moreover, an instruction-set-semantics developer should specify only the abstract syntax and a concrete operational semantics of the language to be analyzed—each analyzer should be generated automatically from this specification.
- Concrete syntactic issues—including (i) decoding (machine code to abstract syntax), (ii) encoding (abstract syntax to machine code), (iii) parsing assembly (assembly code to abstract syntax), and (iv) assembly pretty-printing (abstract syntax to assembly code)—should be handled separately from the abstract syntax and concrete semantics.[1]

[1] The translation of the concrete syntaxes to and from abstract syntax is handled by a generator tool that is separate from TSL, and will not be discussed in this thesis. The relationship between the two systems is similar to that between Flex and Bison. With Flex and Bison, a Flex-generated lexer passes tokens to a Bison-generated parser. In our case, the TSL-defined abstract syntax serves as the formalism for communicating values—namely, instructions' abstract syntax trees—between the two tools.

- There should be a clean interface for analysis developers to specify the abstract semantics for each analysis. An abstract semantics consists of an *interpretation*: an abstract domain and a set of abstract operators (i.e., for the operations of TSL).
- The abstract semantics for each analysis should be separated from the languages to be analyzed so that one does not need to specify multiple versions of an abstract semantics for multiple languages.

Each of these objectives has been achieved in the TSL system: The TSL system translates the TSL specification of each instruction set to a common intermediate representation (CIR) that can be used to create multiple analyzers (§3.1). Each analyzer is specified at the level of the meta-language (i.e., by reinterpreting the operations of TSL), which—by extension to TSL expressions and functions—provides the desired reinterpretation of the instructions of an instruction set (§3.3).

Other notable aspects of our work include

- ***Support for Multiple Analysis Types.*** The system supports several analysis types:
  - Classical worklist-based value-propagation analyses.
  - Transformer-composition-based analyses [74, 169], which are particularly useful for context-sensitive interprocedural analysis, and for relational analyses.
  - Unification-based analyses for flow-insensitive interprocedural analysis.

  In addition, an emulator (for the concrete semantics) is also created.
- ***Implemented Analyses.*** These mechanisms have been instantiated for a number of specific analyses that are useful for analyzing low-level code, including value-set analysis [38, 41] (§3.3.1), affine-relation analysis [38, §7.2] (§3.3.2), def-use analysis (for memory, registers, and flags) (§3.3.3), aggregate structure identification [42] (§3.3.4), and generation of symbolic expressions for an instruction's semantics (§3.3.5).
- ***Established Applicability.*** The capabilities of our approach have been demonstrated by writing specifications for IA32 and PowerPC. These are nearly complete specifications of the integer subset of these languages, and include such features as (1) aliasing among 8-, 16-, and 32-bit registers, e.g., al, ah, ax, and eax (for IA32), (2) endianness, (3) issues arising due

to bounded-word-size arithmetic (overflow/underflow, carry/borrow, shifting, rotation, etc.), and (4) setting of condition codes (and their subsequent interpretation at jump instructions).

The TSL-generated analysis components for IA32 and PowerPC have been put together to create a system that essentially duplicates CodeSurfer/x86 [5] and creates CodeSurfer/ppc32, respectively. We have also experimented with sufficiently complex features of other low-level languages (e.g., register windows for Sun SPARC and conditional execution of instructions for ARM) to know that they fit our specification and implementation models.

The remainder of this chapter is organized as follows: §3.1 presents the overview of the TSL system both from the perspective of instruction-set specifiers (ISS) (§3.1.1) and that of analysis developers (§3.1.2). The section also discusses quirky features of several instruction sets, and discusses how those features are handled in TSL. §3.2 discusses how the TSL compiler generates a CIR from a TSL specification and how the CIR is used for creating analysis components. The section also describes how the TSL system handles some important issues, such as recursion and conditional branches in the CIR. §3.3 presents several analysis components that have been instantiated for developing a system for analyzing low-level code. §3.4 discusses the measure of success and the leverage that the TSL system provides. §3.5 discusses related work.

3.1 Overview of the TSL System

The key principle of the TSL system is the separation of the semantics of a subject language from the analysis semantics in the development of an analysis component. As discussed in §1.4.1, the TSL system is based on semantic reinterpretation, which was originally proposed as a convenient *methodology* for formulating abstract interpretations [73, 110, 134, 144, 148] (see §1.4.1). Semantic reinterpretation involves refactoring the specification of the concrete semantics of a language into two parts: (i) a *client* specification, and (ii) a semantic *core*. The interface to the core consists of certain basetypes, function types, and operators (sometimes called a *semantic algebra* [140]). The client is expressed in terms of this interface. Such an organization permits the core to be *reinterpreted* to produce an alternative semantics for the subject language.

The key insight behind the TSL system is that if a rich enough *meta-language* is provided for writing semantic specifications, one can avoid the *ad hoc* refactoring step. The advantage of this approach is that it allows the TSL system to act as a "YACC-like" tool for generating analysis components from a semantic description of an instruction set.

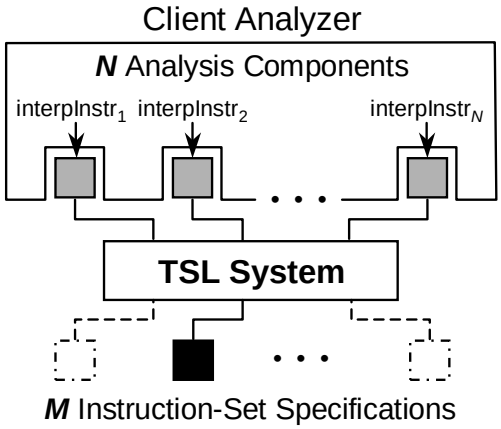


Figure 3.1 The interaction between the TSL system and a client analyzer. The grey boxes represent TSL-generated analysis components.

The TSL system has two classes of users: (1) instruction-set specifiers (ISS) and (2) analysis developers. The former use the TSL language to specify the concrete semantics of different instruction sets (the lower part of Fig. 3.1); the latter use semantic reinterpretation to create new analyses (the upper part of Fig. 3.1). §3.1.1 and §3.1.2 present the TSL system from an instruction-set specifier's standpoint and an analysis developer's standpoint, respectively.

3.1.1 TSL from an ISS's Standpoint

Fig. 3.2 shows part of a specification of the IA32 instruction set taken from the Intel manual [17]. The specification describes the syntax and the semantics of each instruction only in a semi-formal way (i.e., a mixture of English and pseudo-code).

Our work is based on completely formal specifications that are written in a language that we designed (TSL). TSL is a strongly typed, first-order functional language. TSL supports a fixed set of base-types; a fixed set of arithmentic, bitwise, relational, and logical operators; the ability to define

General Purpose Registers:
EAX,EBX,ECX,EDX,ESP,EBP,ESI,EDI,EIP
Each of these registers also has 16- or 8-bit subset names.
Addressing Modes: [sreg:][offset][([base][,index][,scale])]
EFLAGS register: ZF,SF,OF,CF,AF,PF, . . .
	ADD r/m32,r32; Add r32 to r/m32
	ADD r/m16,r16; Add r16 to r/m16 . . .
	Operation: DEST ← DEST + SRC;
	Flags Affected: The OF,SF,ZF,AF,CF, and PF flags are set according to the result.

Figure 3.2 A part of the Intel manual's specification of IA32's `add` instruction.

recursive data-types, map-types, and user-defined functions; and a mechanism for deconstruction by means of pattern matching.

Basetypes. Fig. A.1 shows the basetypes that TSL provides. There are two categories of primitive base-types: *unparameterized* and *parameterized*. An unparameterized base-type is just a set of terms. For example, BOOL is a type consisting of truth values, INT32 is a type consisting of 32-bit signed whole numbers, etc. MAP[α, β] is a predefined parameterized type, with parameters α and β. Each of the following is an instance of the parameterized type MAP:

```
MAP[INT32,INT8]
MAP[INT32,BOOL]
MAP[INT32,MAP[INT8,BOOL]]
```

TSL supports arithmetic/logical operators ($+$, $-$, $*$, $/$, !, $\&\&$, $||$, xor), bit-manipulation operators ($\sim$, $\&$, $|$, $\hat{}$, $\ll$, $\gg$, right-rotate, left-rotate), relational operators ($<$, $<=$, $>$, $>=$, $==$, !=), and a conditional-expression operator ($?:$). TSL also provides access/update operators for map-types.

More details of the TSL syntax and semantics can be found in Appendix A.

Specifying an Instruction Set. Fig. 3.4(a) shows a snippet of the TSL specification that corresponds to Fig. 3.2.[2] Much of what an instruction-set specifier writes in a TSL specification is similar to writing an interpreter for an instruction set in first-order ML [99]. One specifies (i) the abstract-syntax grammar of the instruction-set, (ii) a type for concrete states, and (iii) the concrete semantics of each instruction.

[2]The TSL specification is simplified to make the presentation simpler.

Type	Terms	Constants
BOOL	false, true	false, true
INT64	64-bit signed integers	0d64, 1d64, 2d64, ...
INT32	32-bit signed integers	0d32, 1d32, 2d32, ...
INT16	16-bit signed integers	0d16, 1d16, 2d16, ...
INT8	8-bit signed integers	0d8, 1d8, 2d8, ...
STR	Sequences of characters. All characters except '\000' permitted.	"" "ab...AB...01...!%..." "\n\r\b\t\f\'\"\\" "\001\002\003..."
MAP[α,β]	Maps	*no constants*

Figure 3.3 Syntax of constants of primitive type.

Reserved, but User-Defined Types and Reserved Functions. Each specification must define several reserved (but user-defined) types: instruction (lines 2–9 of Fig. 3.4(a)); state—e.g., for 32-bit Intel x86 the type *state* is a triple of maps (lines 10–12 of Fig. 3.4(a)); as well as the reserved TSL function interpInstr (lines 17–30 of Fig. 3.4(a)). These reserved types and functions form part of the API available to *analysis engines* that use the TSL-generated transformers (CIR).

The definition of types and constructors on lines 2–9 of Fig. 3.4(a) is an abstract-syntax grammar for IA32. Type reg consists of nullary constructors for IA32 registers, such as EAX() and EBX(); *flag* consists of nullary constructors for the IA32 condition codes, such as ZF() and SF(). Lines 4–6 define types and constructors to represent the various kinds of operands that IA32 supports, i.e., various sizes of immediate, direct register, and indirect memory operands. The reserved (but user-defined) type instruction consists of user-defined constructors for each instruction, such as MOV and ADD.

The type state specifies the structure of the execution state. The state for IA32 is defined on lines 10–12 of Fig. 3.4(a) to consist of three maps, i.e., a memory-map, a register-map, and a flag-map. The *concrete semantics* is specified by writing a function named interpInstr (see lines 17–30

```
[1]  // User-defined abstract syntax
[2]  reg: EAX() | EBX() | . . . ;
[3]  flag: ZF() | SF() | . . . ;
[4]  operand: Indirect(reg reg INT8 INT32)
[5]              | DirectReg(reg)
[6]              | Immediate(INT32) | ...;
[7]   instruction
[8]       : MOV(operand operand)
[9]       | ADD(operand operand| . . . ;
[10] state: State(MAP[INT32,INT8]   // memory-map
[11]              MAP[reg32,INT32]  // register-map
[12]              MAP[flag,BOOL]);   // flag-map
[13] // User-defined functions
[14] INT32 interpOp(state S, operand op) { . . . };
[15] state updateFlag(state S, . . . ) { . . . };
[16] state updateState(state S, . . . ) { . . . };
[17] state interpInstr(instruction I, state S) {
[18]   with(I) (
[19]     MOV(dstOp, srcOp):
[20]       let srcVal = interpOp(S, srcOp);
[21]       in ( updateState( S, dstOp, srcVal ) ),
[22]     ADD(dstOp, srcOp):
[23]       let dstVal = interpOp(S, dstOp);
[24]           srcVal = interpOp(S, srcOp);
[25]           res = dstVal + srcVal;
[26]           S2 = updateFlag(S, dstVal, srcVal, res);
[27]       in ( updateState( S2, dstOp, res ) ),
[28]     . . .
[29]   );
[30] };
```

```
[1]  template <class BT> class CIR {
[2]    class reg { . . . };
[3]    class EAX : public reg { . . . }; . . .
[4]    class flag { . . . };
[5]    class ZF : public flag { . . . }; . . .
[6]    class operand { . . . };
[7]    class Indirect: public operand { . . . }; . . .
[8]    class instruction { . . . };
[9]    class MOV : public instruction { . . .
[10]     operand  op1; operand  op2; . . .
[11]   };
[12]   class MOV : public instruction { . . . }; . . .
[13]   class state { . . . };
[14]   class State: public state { . . . };
[15]   BT::INT32 interpOp(state S, operand op) { . . . };
[16]   state updateFlag(state S, . . . ) { . . . };
[17]   state updateState(state S, . . .) { . . . };
[18]   state interpInstr(instruction I, state S) {
[19]    switch(I.id) {
[20]      case ID_MOV: . . .
[21]      case ID_ADD:
[22]        operand dstOp = I.get_child1();
[23]        operand srcOp = I.get_child2();
[24]        BT::INT32 dstVal = interpOp(S, dstOp);
[25]        BT::INT32 srcVal = interpOp(S, srcOp);
[26]        BT::INT32 res = BT::Plus(dstVal, srcVal);
[27]        state S2 = updateFlag(S, dstVal, srcVal, res);
[28]        ans = updateState( S2, dstOp, res );
[29]        break;
[30]    . . . }
[31]   }};
```

Figure 3.4 (a) A part of the TSL specification of IA32 concrete semantics, which corresponds to the specification of add from the IA32 manual. Reserved types and function names are underlined, (b) A part of the CIR generated from (a); The CIR is simplified in this presentation.

of Fig. 3.4(a)), which maps an instruction and a state to a state. For instance, the semantics of ADD is to evaluate the two operands in the input state S and create a return state in which the target location holds the summation of the two values and the flags hold appropriate flag values.

3.1.1.1 Case Study of Instruction Sets

In this section, we discuss the quirky characteristics of some instruction sets, and various ways these can be handled in TSL.

IA32. To provide compatibility with 16-bit and 8-bit versions of the instruction set, IA32 provides overlapping register names, such as AX (the lower 16-bits of EAX), AL (the lower 8-bits of AX), and AH (the upper 8-bits of AX). There are two possible ways to specify this feature in TSL. One is to keep three separate maps, for 32-bit registers, 16-bit registers, and 8-bit registers, respectively, and specify that updates to any one of the maps affect the other two maps. Another is to keep one 32-bit map for registers, and obtain the value of a 16-bit or 8-bit register by masking the value of the 32-bit register. (The former can yield more precise VSA results.)

Another characteristic to note is that IA32 keeps condition codes in a special register, called EFLAGS.[3] One way to specify this feature is to declare "reg32: Eflags();", and make every flag manipulation fetch the bit value from an appropriate bit position of the value associated with Eflags in the register-map. Another way is to have symbolic flags, as in our examples, and have every manipulation of EFLAGS affect the entries in a flag-map for the individual flags.

ARM. Almost all ARM instructions contain a condition field that allows an instruction to be executed conditionally, depending on condition-code flags. This feature reduces branch overhead and compensates for the lack of a branch predictor. However, it may worsen the precision of an abstract analysis because in most instructions' specifications, the abstract values from two arms of a TSL conditional expression would be joined.

[3] Many other instruction sets, such as SPARC, PowerPC, and ARM, also use a special register to store condition codes.

```
[1]  MOVEQ(destReg, srcOprnd):
[2]    let cond = flagMap(EQ());
[3]        src = interpOperand(curState, srcOprnd);
[4]        a = regMap[destReg |-> src];
[5]        b = regMap;
[6]        answer = cond ? a : b;
[7]    in ( answer )
```

Figure 3.5 An example of the specification of an ARM conditional-move instruction in TSL.

For example, MOVEQ is one of ARM's conditional instructions; if the flag EQ is true when the instruction starts executing, it executes normally; otherwise, the instruction does nothing. Fig. 3.5 shows the specification of the instruction in TSL. In many abstract semantics, the conditional expression "*cond* ? *a* : *b*" will be interpreted as a join of the original register map *b* and the updated map *a*, i.e., *join*(*a*,*b*). Consequently, *destReg* would receive the join of its original value and *src*, even when *cond* is known to have a definite value (TRUE or FALSE) in VSA semantics. The paired-semantics mechanism presented in §3.2.3 can help with improving the precision of analyzers by avoiding joins. When the CIR is instantiated with a paired semantics of VSA_INTERP and DUA_INTERP, and the VSA value of *cond* is FALSE, the DUA_INTERP value for *answer* gets empty *def*- and *use*-sets because the true branch *a* is known to be unreachable according to the VSA_INTERP value of *cond* (instead of non-empty sets for *def*s and *use*s that contain all the definitions and uses in *destReg* and *srcOprnd*).

SPARC. SPARC uses register windows to reduce the overhead associated with saving registers to the stack during a conventional function call. Each window has 8 in, 8 out, 8 local, and 8 global registers. Outs become ins on a context switch, and the new context gets a new set of out and local registers. A specific platform will have some total number of registers, which are organized as a circular buffer; when the buffer becames full, registers are spilled to the stack to free up a sufficient number for the called procedure. Fig. 3.6 shows a way to accomodate this feature. The syntactic

```
[1]  reg32 : Reg(INT8) | CWP() | . . .;
[2]  reg32 : OutReg(INT8) | InReg(INT8) | . . .;
[3]  state: State( . . . , MAP[var32,INT32], . . . );
[4]  INT32 RegAccess(MAP[var32,INT32] regmap, reg32 r) {
[5]   let cwp = regmap(CWP());
[6]      key = with(r) (
[7]       OutReg(i):
[8]          Reg(8+i+(16+cwp*16)%(NWINDOWS*16),
[9]       InReg(i): Reg(8+i+cwp*16),
[10]    . . . );
[11] in ( regmap(key) )
[12]}
```

Figure 3.6 A method to handle the SPARC register window in TSL.

register (OutReg(n) or InReg(n), defined on line 2) in an instruction is used to obtain a semantic register (Reg(m), defined on line 1, where m represents the register's global index), which is the key used for accesses on and updates to the register map. The desired index of the semantic register is computed from the index of the syntactic register, the value of CWP (the current window pointer) from the current state, and the platform-specific value NWINDOWS (lines 8–9).

3.1.1.2 Common Intermediate Representation (CIR)

Fig. 3.4(b) shows part of the common intermediate representation (CIR) generated by the TSL compiler from Fig. 3.4(a).[4] The CIR generated for a given TSL specification is a C++ template that can be used to create multiple analysis components by instantiating the template with different semantic reinterpretations. Each generated CIR is *specific* to a given instruction-set specification, but *common* (whence the name CIR) across generated analyses.

[4]This CIR has been simplified for the presentation in the thesis.

Each generated CIR is a template class that takes as input class BT (standing for base-type interpretation), which is an abstract domain for an analysis (line 1 of Fig. 3.4(b)). The user-defined abstract syntax (lines 2–9 of Fig. 3.4(a)) is translated to a set of C++ abstract-syntax classes (lines 2–12 of Fig. 3.4(b)). The user-defined types, such as reg, operand, and instruction, are translated to abstract C++ classes, and the constructors, such as EAX(), Indirect(_,_,_,_), and ADD(_,_), are subclasses of the appropriate parent abstract C++ classes.

Each user-defined function is translated to a CIR function (lines 15–31 of Fig. 3.4(b)). Each TSL basetype and basetype-operator is prepended with the template parameter name BT; BT is supplied by an analysis developer for the analysis of interest. The with expression and the pattern matching on lines 18–22 of Fig. 3.4(a) are translated into switch statements in C++ (lines 19–30 in Fig. 3.4(b)).

With-normalization. The TSL front-end performs *with-normalization*, which transforms all multi-level with expressions to use only one-level patterns, and then compiles the one-level pattern via the pattern-compilation algorithm developed by M. Pettersson [153, 178]. The algorithm for compiling term pattern-matching for functional languages is inspired by finite automata theory. The algorithm avoids duplicating code and introducing redundant or sub-optimal discrimination tests by viewing patterns as regular expressions and optimizing the finite automaton that is built to recognize them.

The function calls for obtaining the values of the two operands (lines 23–24 in Fig. 3.4(a)) correspond to the C++ code on lines 22–25 in Fig. 3.4(b). The TSL basetype-operator + on line 25 in Fig. 3.4(a) is translated into a call to BT::Plus, as shown on line 26 in Fig. 3.4(b). The function calls for updating the state (lines 26–27 in Fig. 3.4(a)) are translated into C++ code (lines 27–28 in Fig. 3.4(b)).

§3.2 presents more details as to how CIR is generated and what kind of facilities CIR provides for creating analysis components.

3.1.2 TSL from an Analysis Developer's Standpoint

An analysis developer creates a new analysis component by (i) redefining (in C++) the TSL basetypes (BOOL, INT32, INT8, etc.), and (ii) redefining (in C++) the primitive operations on basetypes ($+_{\mathsf{INT32}}$, $+_{\mathsf{INT8}}$, etc.). These are used to instantiate the CIR template by passing a class of basetypes as the template parameter. This implicitly defines an alternative interpretation of each expression and function in an instruction-set's concrete semantics (including interpInstr), and thereby yields an alternative semantics for an instruction set from its concrete semantics.

Tab. 3.1 shows the implementations of primitives for three selected analyses: value-set analysis (VSA, see §3.3.1), def-use analysis (DUA, see §3.3.3), and quantifier-free bit-vector semantics (QFBV, see §3.3.5). Each interpretation defines an abstract domain. For example, line 3 of each column defines the abstract-domain class for INT32: ValueSet32, UseSet, and QFBVTerm32. To define an interpretation, one needs to define 42 basetype operators, most of which have four variants, for 8-, 16-, 32-, and 64-bit integers, as well as 12 map *access/update* operations. Each abstract domain is also required to contain a set of reserved functions, such as *join*, *meet*, and *widen*, which forms an additional part of the API available to analysis engines that use TSL-generated transformers (see §3.3).

Usage of TSL-Generated Analysis Components. Fig. 3.7 shows how the CIR is connected to an analysis solver. The analysis solver in Fig. 3.7 uses classical worklist-based value propagation in which the TSL-generated transformer interpInstr is invoked with an instruction and the current state S. On each iteration of the main loop of the solver, changes (new_S) are propagated to successors/predecessors (depending on propagation direction). §3.3 summarizes three kinds of analysis engines including worklist-based value propagation.

Generated Transformers. Consider the instruction "add ebx, eax", which causes the sum of the values of the 32-bit registers ebx and eax to be assigned into ebx. When Fig. 3.4(b) is instantiated with the three interpretations from Tab. 3.1, lines 17–30 of Fig. 3.4(a) implement the three transformers that are presented (using mathematical notation) in Tab. 3.2.

Table 3.1 Parts of the declarations of the basetypes, basetype-operators, and map-access/update functions for three analyses.

VSA	DUA	QFBV
[1] class VSA_INTERP {	[1] class DUA_INTERP {	[1] class QFBV_INTERP {
[2] // basetype	[2] // basetype	[2] // basetype
[3] typedef ValueSet32 INT32;	[3] typedef UseSet INT32;	[3] typedef QFBVTerm32 INT32;
[4] . . .	[4] . . .	[4] . . .
[5] // basetype-operators	[5] // basetype-operators	[5] // basetype-operators
[6] INT32 Add(INT32 a, INT32 b) {	[6] INT32 Add(INT32 a, INT32 b) {	[6] INT32 Add(INT32 a, INT32 b) {
[7] return a.addValueSet(b);	[7] return a.Union(b);	[7] return QFBVPlus32(a, b);
[8] }	[8] }	[8] }
[9] . . .	[9] . . .	[9] . . .
[10] // map-basetypes	[10] // map-basetypes	[10] // map-basetypes
[11] typedef Dict<reg32,INT32>	[11] typedef Dict<var32,INT32>	[11] typedef QFBVArray
[12] REGMAP32;	[12] REGMAP32;	[12] REGMAP32;
[13] . . .	[13] . . .	[13] . . .
[14] // map-access/update functions	[14] // map-access/update functions	[14] // map-access/update functions
[15] INT32 MapAccess(	[15] INT32 MapAccess(	[15] INT32 MapAccess(
[16] REGMAP32 m, reg32 k) {	[16] REGMAP32 m, reg32 k) {	[16] REGMAP32 m, reg32 k) {
[17] return m.Lookup(k);	[17] return m.Lookup(k);	[17] return QFBVArrayAccess(m,k);
[18] }	[18] }	[18] }
[19] REGMAP32	[19] REGMAP32	[19] REGMAP32
[20] MapUpdate(REGMAP32 m,	[20] MapUpdate(REGMAP32 m,	[20] MapUpdate(REGMAP32 m,
[21] reg32 k, INT32 v) {	[21] reg32 k, INT32 v) {	[21] reg32 k, INT32 v) {
[22] return m.Insert(k, v);	[22] return m.Insert(k,v);	[22] return QFBVArrayUpdate(m,k,v);
[23] }	[23] }	[23] }
[24] . . .	[24]. . .	[24] . . .
[25]};	[25]};	[25]};

Table 3.2 Transformers generated by the TSL system.

Analysis	Generated Transformers for "add ebx, eax"
1.VSA	λS.S[ebx $\mapsto$ S(ebx)$+^{vsa}$S(eax)] [ZF $\mapsto$ (S(ebx)$+^{vsa}$S(eax) = 0)][*more flag updates*]
2.DUA	[ebx $\mapsto$ {eax, ebx}, ZF $\mapsto$ {eax, ebx}, . . .]
3.QFBV	(ebx′ = ebx$+^{32}$eax) $\wedge$ (ZF′ $\Leftrightarrow$ (ebx$+^{32}$eax = 0)) $\wedge$ (SF′ $\Leftrightarrow$ (ebx$+^{32}$eax$<$ 0)) $\wedge$ . . .

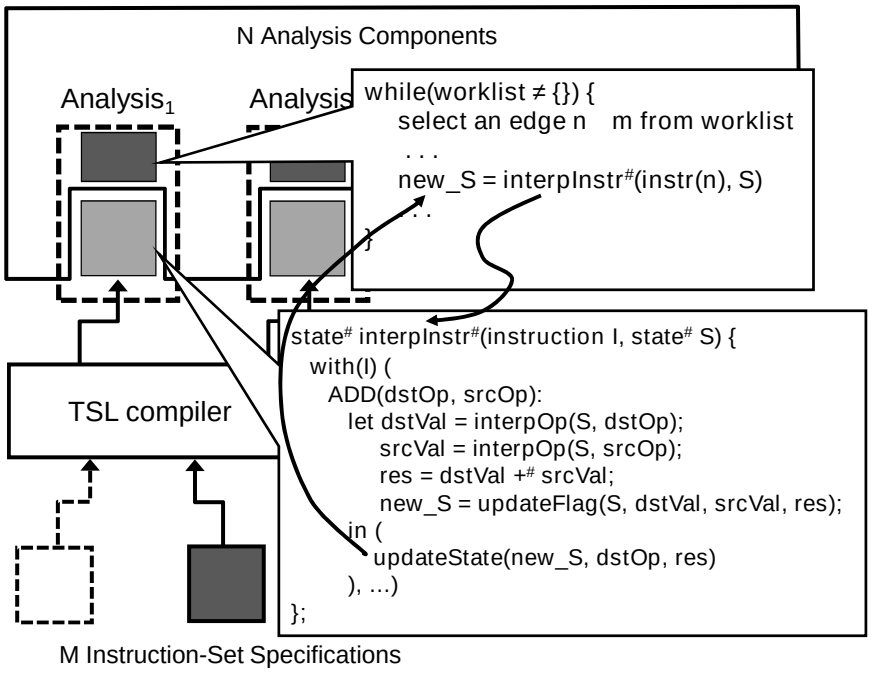


Figure 3.7 How a TSL-generated analysis component (interpInstr$^\sharp$) is invoked in a solver that uses classical worklist-based value propagation.

3.2 Various Aspects of a Common Intermediate Representation

Given a TSL specification of an instruction set, the TSL system generates a CIR that consists of two parts: one is a list of C++ classes for the user-defined abstract-syntax grammar; the other is a list of C++ template functions for the user-defined functions, including the interface function interpInstr. The C++ functions are generated by linearizing the TSL specification, in evaluation order, into a series of C++ statements as described in §3.1.1.2.

However, there are some important issues that need to be properly handled for the resulting code to be able to be used to create abstract interpreters for an instruction-set specification. In particular, the code generated for each transformer must be able to: (i) execute over abstract states (§3.2.2), (ii) possibly propagate abstract states to more than one successor in a conditional expression (§3.2.2.1), (iii) compare abstract states and terminate abstract execution when a fixed point is reached (§3.2.2.2), and (iv) apply widening operators, if necessary, to ensure termination (§3.2.2.2).

In §3.2.1, we discuss an additional issue that arises in CIR generation, which is important for avoiding loss of precision for some generated analyzers. §3.2.3 presents the *paired-semantics* facility that the TSL system provides.

3.2.1 Two-Level CIR

The examples given in Fig. 3.4(b), Fig. 3.10, and Fig. 3.11(b), show slightly simplified versions of CIR code. The TSL system actually generates CIR code in which all the basetypes, basetype-operators, and *access/update* functions are appended with one of two predefined namespaces that define a *two-level* interpretation [111, 149]: CONCINTERP for concrete interpretation (i.e., interpretation in the concrete semantics), and ABSINTERP for abstract interpretation. Either CONCINTERP or ABSINTERP would replace the occurrences of BT in the example CIR shown in Fig. 3.4(b), Fig. 3.10, and Fig. 3.11(b).

The reason for using a two-level CIR is that the specification of an instruction set often contains some manipulations of values that should always be treated as concrete values. For example, an instruction-set specification developer could follow the approach taken in the PowerPC manual

```
[1]  // User-defined abstract-syntax grammar
[2]  instruction: . . .
[3]     | BCx(BOOL BOOL INT32 BOOL BOOL)
[4]     | . . . ;
[5]  // User-defined functions
[6]  state interpInstr(instruction I, state S) {
[7]     . . .
[8]     BCx(BO, BI, target, AA, LK):
[9]       let . . .
[10]          cia = RegValue32(S, CIA()); // current address
[11]          new_ia = (AA ? target          // direct: BCA/BCLA
[12]                       : cia + target); // relative: BC/BCL
[13]          lr = RegValue32(S, LR());    // linkage address
[14]          new_lr =
[15]            (LK ? cia + 4 // change the link register: BCL/BCLA
[16]                : lr); // do not change the link register: BC/BCA
[17]     . . .
[18]}
```

Figure 3.8 A fragment of the PowerPC specification for interpreting BCx instructions (BC, BCA, BCL, BCLA).

[27] and specify variants of the conditional branch instruction (BC, BCA, BCL, BCLA) of PowerPC by interpreting some of the fields in the instruction (AA and LK) to determine which of the four variants is being executed (Fig. 3.8).

Another reason that this issue arises is that most well-designed instruction sets have many regularities, and it is convenient to factor the TSL specification to take advantage of these regularities when specifying the semantics. Such factoring leads to shorter specifications, but leads to the introduction of auxiliary functions in which one of the parameters holds a constant value for a *given* instruction. Fig. 3.9 shows an example of factoring. The IA32 instructions `add` and `sub` both have two operands and can share the code for fetching the values of the two operands. Lines 4–5 are

```
[1]  AddSubInstr(op, dstOp, srcOp): // ADD or SUB
[2]    let dstVal = interpOp(S, dstOp);
[3]        srcVal = interpOp(S, srcOp);
[4]        ans = (op == ADD() ? dstVal + srcVal
[5]                           : dstVal - srcVal); // SUB()
[6]    in ( . . . ),
[7]  . . .
```

Figure 3.9 An example of factoring in TSL.

the instruction-specific operations; the equality expression "op == ADD()" on line 4 can be (and should be) interpreted in concrete semantics.

In both cases, the precision of an abstract transformer can sometimes be improved—and is never made worse—by interpreting subexpressions associated with the manipulation of concrete values in concrete semantics. For instance, consider a TSL expression $let\ v\ =\ (b\ ?\ 1\ :\ 2)$ that occurs in a context in which b is definitely a concrete value; v will get a precise value—either 1 or 2—when b is concretely interpreted. However, if b is not expressible precisely in a given abstract domain, the conditional expression "$(b\ ?\ 1\ :\ 2)$" will be evaluated by joining the two branches, and v will not hold a precise value. (It will hold the abstraction of $\{1, 2\}$.)

Binding-time analysis. To address the issue, we perform binding-time analysis [109] on the TSL code, the outcome of which is that expressions associated with the manipulation of concrete values in an instruction are annotated with C, and others with A. We then generate the two-level CIR by appending CONCINTERP for C values, and ABSINTERP for A values. The generated CIR is instantiated for an analysis transformer by defining ABSINTERP. The TSL translator supplies a predefined concrete interpretation for CONCINTERP.

The instruction-set-specification developer annotates the top-level user-defined (but reserved) functions, including interpInstr, with binding-time information.

EXPORT $<A>$ interpInstr($<C>$, $<A>$)

The first argument of type instruction of interpInstr is annotated with $<C>$, which indicates that all the data extracted from the instruction are treated as *concrete*; the second argument of type state of interpInstr is annotated with $<A>$, which indicates that all the data extracted from the state are treated as *abstract*. The return type is also annotated as abstract. The binding-time information $<A>$ is propagated to the caller-sites of interpInstr.

More details of the TSL syntax for binding-time analysis can be found in Appendix A.

3.2.2 Execution Over Abstract States

There are (at least) four issues that arise: during the abstract interpretation of each transformer, the abstract interpreter must be able to (i) execute over abstract states, (ii) execute both branches of a conditional expression, (iii) compare abstract states and terminate abstract execution when a fixed point is reached, and (iv) apply widening operators, if necessary, to ensure termination. The following subsections discuss how these issues are handled in the translation to CIR.

3.2.2.1 Conditional Expressions

Fig. 3.10 shows part of the CIR that corresponds to the TSL expression "let answer = a ? b : c". Bool3 is an abstract domain of Booleans (which consists of three values {FALSE, MAYBE, TRUE}, where MAYBE means "may be FALSE or may be TRUE"). The TSL conditional expression is translated into three if-statements (lines 3–7, lines 8–12, and lines 13–15 in Fig. 3.10). The body of the first if-statement is executed when the Bool3 value for a is possibly false (i.e., either FALSE or MAYBE). Likewise, the body of the second if-statement is executed when the Bool3 value for a is possibly true (i.e., either TRUE or MAYBE). The body of the third if-statement is executed when the Bool3 value for a is MAYBE. Note that in the body of the third if-statement, answer is overwritten with the *join* of t1 and t2 (line 14).

The Bool3 value for the translation of a TSL BOOL-valued value is fetched by getBool3Value, which is one of the TSL interface functions that each interpretation is required to define for the type BOOL. Each analysis developer decides how to handle conditional branches by defining getBool3Value. It is always sound for getBool3Value to be defined as the constant function that

```
[1]   BT::BOOL t0 = . . . ; // translation of a
[2]   BT::INT32 t1, t2, answer;
[3]   if(Bool3::possibly_false(t0.getBool3Value())) {
[4]       . . .
[5]       t1 = . . . ; // translation of b
[6]       answer = t1;
[7]   }
[8]   if(Bool3::possibly_true(t0.getBool3Value())) {
[9]       . . .
[10]      t2 = . . . ; // translation of c
[11]      answer = t2;
[12] }
[13] if(t0.getBool3Value() == Bool3::MAYBE) {
[14]      answer = t1.join(t2);
[15] }
```

Figure 3.10 The translation of the conditional expression "let answer = a ? b : c".

always returns MAYBE. For instance, this constant function is useful when Boolean values cannot be expressed in an abstract domain, such as DUA for which the abstract domain for BOOL is a set of *use*s. For an analysis where Bool3 is itself the abstract domain for type BOOL, such as VSA, getBool3Value returns the Bool3 value from evaluating the translation of a so that either an appropriate branch or both branches can be abstractly executed.

3.2.2.2 Comparison, Termination, and Widening

Recursion is not often used in TSL specifications, but is needed for handling some instructions that involve iteration, such as the IA32 string-manipulation instructions (STOS, LODS, MOVS, etc., with various REP prefixes), and the PowerPC multiple-word load/store instructions (LMW, STMW, etc). For these instructions, the amount of work performed is controlled either by the value of a register, the value of one or more strings, etc. These instructions can be specified in TSL using

```
[1]  state repMovsd(state S, INT32 count) {
[2]    count == 0
[3]      ? S
[4]      : with(S) (
[5]          State(mem, regs, flags):
[6]            let direction = flags(DF());
[7]                edi = regs(EDI());
[8]                esi = regs(ESI());
[9]                src = MemAccess_32_8_LE_32(mem, esi);
[10]               newRegs = direction
[11]                  ? regs[EDI()|->edi-4][ESI()|->esi-4]
[12]                  : regs[EDI()|->edi+4][ESI()|->esi+4]
[13]               newMem = MemUpdate_32_8_LE_32(
[14]                  memory, edi, src);
[15]               newS = State(newMem, newRegs, flags);
[16]           in ( repMovsd(newS, count - 1) )
[17]     )
[18]};
```

```
[1]  state global_S;
[2]  BT::INT32 global_count;
[3]  state global_retval;
[4]  BT::state repMovsd(
[5]       INTERP::state S, BT::INT32 count) {
[6]       global_S = ⊥;
[7]       global_count = ⊥;
[8]       global_retval = ⊥;
[9]       return repMovsdAux(S, count);
[10]};
[11]INTERP::state repMovsdAux(
[12]    INTERP::state S, BT::INT32 count) {
[13]    // Widen and test for convergence
[14]    state tmp_S = global_S ▽ (global_S ⊔ S);
[15]    BT::INT32 tmp_count =
[16]          global_count ▽ (global_count ⊔ count);
[17]    if(tmp_S ⊑ global_S
[18]       && tmp_count ⊑ global_count) {
[19]       return global_retval;
[20]    }
[21]    S = tmp_S; global_S = tmp_S;
[22]    count = tmp_count; global_count = tmp_count;
[23]
[24]    // translation of the body of repMovsd
[25]    . . .
[26]    state newS = . . . ;
[27]    state t = repMovsdAux(newS, count - 1);
[28]    global_retval = global_retval ⊔ t;
[29]    return global_retval;
[30]};
```

Figure 3.11 (a) A recursive TSL function, (b) The translation of the recursive function from (a). For simplicity, some mathematical notation is used, including ⊔ (join), ▽ (widening), ⊑ (approximation), and ⊥ (bottom).

recursion.[5] For each recursive function specified by an instruction-set specification developer, the TSL system generates a function that appropriately compares abstract values and terminates the recursion if abstract values are found to be equal (i.e., the recursion has reached a fixed point). The function is also prepared to apply the widening operator that the analysis developer has specified for the abstract domain in use.

For example, Fig. 3.11(a) shows the user-defined TSL function that handles "rep movsd", which copies the contents of one area of memory to a second area.[6] The amount of memory to be copied is passed into the function as the argument count. Fig. 3.11(b) shows its translation into the CIR. A recursive function like repMovsd (Fig. 3.11(a)) is automatically split by the TSL compiler into two functions, repMovsd (line 4 of Fig. 3.11(b)) and repMovsdAux (line 11 of Fig. 3.11(b)). The TSL system initializes appropriate global variables global_S and global_count (lines 6–8) in repMovsd, and then calls repMovsdAux (line 9). At the beginning of repMovsdAux, it generates statements that widen each of the global variables with respect to the arguments, and test whether all of the global variables have reached a fixpoint (lines 13–17). If so, repMovsdAux returns global_retval (line 19). If not, the body of repMovsdAux is analyzed again (lines 24–27). Note that at the translation of each normal return from repMovsdAux (e.g., line 28), the return value is joined into global_retval. The TSL system requires each analysis developer to define the functions *join* and *widen* for the basetypes of the interpretation used in the analysis.

3.2.3 Paired Semantics

Our system allows easy instantiations of *reduced product*s [74] by means of *paired semantics*. The TSL system provides a template for paired semantics as shown in Fig. 3.12(a).

The CIR is instantiated with a *paired* semantic domain defined with two interpretations, INTERP1 and INTERP2 (each of which may itself be a paired semantic domain), as shown on line 1 of Fig. 3.12(b). The communication between interpretations may take place in basetype-operators or *access/update* functions; Fig. 3.12(b) is an example of the latter. The two components

[5] Currently, TSL supports only tail-recursion.

[6] repMovsd is called by interpInstr, which passes in the value of register `ecx`, and sets `ecx` to 0 after repMovsd returns.

(a)

```
[1]  template <typename INTERP1, typename INTERP2>
[2]  class PairedSemantics {
[3]    typedef PairedBaseType<INTERP1::INT32, INTERP2::INT32> INT32;
[4]    . . .
[5]    INT32 MemAccess_32_8_LE_32(MEMMAP32_8 mem, INT32 addr) {
[6]      return INT32(INTERP1::MemAccess_32_8_LE_32(mem.GetFirst(), addr.GetFirst()),
[7]                   INTERP2::MemAccess_32_8_LE_32(mem.GetSecond(), addr.GetSecond()));
[8]    }
[9]  };
```

(b)

```
[1]  typedef PairedSemantics<VSA_INTERP, DUA_INTERP> DUA;
[2]  template<> DUA::INT32 DUA::MemAccess_32_8_LE_32(
[3]                                DUA::MEMMAP32_8 mem, DUA::INT32 addr) {
[4]    DUA::INTERP1::MEMMAP32_8 memory1 = mem.GetFirst();
[5]    DUA::INTERP2::MEMMAP32_8 memory2 = mem.GetSecond();
[6]    DUA::INTERP1::INT32 addr1 = addr.GetFirst();
[7]    DUA::INTERP2::INT32 addr2 = addr.GetSecond();
[8]    DUA::INT32 answer = interact(mem1, mem2, addr1, addr2);
[9]    return answer;
[10]}
```

Figure 3.12 (a) A part of the template class for paired semantics; (b) an example of C++ explicit template specialization to create a reduced product.

of the paired-semantics values are deconstructed on lines 4–7 of Fig. 3.12(b), and the individual INTERP1 and INTERP2 components from *both* inputs can be used (as illustrated by the call to *interact* on line 8 of Fig. 3.12(b)) to create the paired-semantics return value, answer. Such overridings of basetype-operators and *access/update* functions are done by C++ explicit specialization of members of class templates (this is specified in C++ by "template<>"; see line 2 of Fig. 3.12(b)).

We also found this method of CIR instantiation to be useful to perform a form of reduced product when analyses are split into multiple phases, as in a tool like CodeSurfer/x86. CodeSurfer/x86 carries out many analysis phases, and the application of its sequence of basic analysis phases is itself iterated. On each round, CodeSurfer/x86 applies a sequence of analyses: VSA, DUA, and several others. VSA is the primary workhorse, and it is often desirable for the information acquired

```
[1]  with(op) ( . . .
[2]    Indirect32(base, index, scale, disp):
[3]     let  addr = base
[4]            + index * SignExtend8To32(scale)
[5]            + disp;
[6]          m = MemUpdate_32_8_LE_32(
[7]                        mem,addr,v);
[8]  . . .)
```

Figure 3.13 A fragment of updateState.

by VSA to influence the outcomes of other analysis phases by pairing the VSA interpretation with another interpretation.

We can use the paired-semantics mechanism to obtain desired *multi-phase interactions* among our generated analyzers—typically, by pairing the VSA interpretation with another interpretation. For instance, with DUA_INTERP alone, the information required to obtain abstract memory location(s) for addr is lost because the DUA basetype-operators (used on $+$ and $*$ on lines 4–5 of Fig. 3.13) just return the union of the arguments' *use* sets. With the pairing of VSA_INTERP with DUA_INTERP (line 1 of Fig. 3.12(b)), DUA can use the abstract address computed for addr2 (line 7 of Fig. 3.12(b)) by VSA_INTERP, which uses VSA_INTERP::Add and VSA_INTERP::Mult; the latter operators operate on a numeric abstract domain (rather than a set-based one).

Note that during the application of the paired semantics, VSA interpretation will be carried out on the VSA component of paired intermediate values. In some sense, this is duplicated work; however, a paired semantics is typically used only in a phase of transformer generation where the transformers are generated during a single pass over the interprocedural CFG to generate a transformer for each instruction. Thus, only a limited amount of VSA evaluation is performed (equal to what would be performed to check that the VSA solution is a fixed point).

3.3 TSL-Generated Analysis Components

In this section, we present various analyses that are created by the TSL system. As illustrated in Fig. 3.7, a version of the interface function interpInstr is created for each analysis. Each analysis engine calls interpInstr at appropriate moments to obtain a transformer for an instruction being processed. Analysis engines can be categorized as follows:

- *Worklist-Based Value Propagation (or Transformer Application)* [TA]. These perform classical worklist-based value propagation in which generated transformers are applied, and changes are propagated to successors/predecessors (depending on propagation direction). Context sensitivity in such analyses is supported by means of the call-string approach [169]. VSA uses this kind of analysis engine (§3.3.1).
- *Transformer Composition* [TC]. These generally perform flow-sensitive, context-sensitive interprocedural analysis. DUA (§3.3.3) uses this kind of analysis engine.
- *Unification-Based Analyses* [UB]. These perform flow-insensitive interprocedural analysis. ASI (§3.3.4) uses this kind of analysis engine.

For each analysis, the CIR is instantiated with an interpretation by an analysis developer. This mechanism provides wide flexibility in how one can couple the system to an external package. One approach, used with VSA, is that the analysis engine (written in C++) calls interpInstr directly. In this case, the instantiated CIR serves as a *transformer evaluator*: interpInstr is prepared to receive an instruction and an abstract state, and return an abstract state. Another approach, used in DUA, is employed when interfacing to an analysis component that has its own input language for specifying abstract transformers. In this case, the instantiated CIR serves as a *transformer generator*: interpInstr is prepared to receive an instruction and a default abstract state[7] and return a transformer specification in the analysis component's input language.

The following subsections discuss how the CIR is instantiated for various analyses.

[7] In the case of transformer generation for a TC analyzer, the default state is the identity function.

3.3.1 Creation of a TA Transformer Evaluator for VSA

VSA is a combined numeric-analysis and pointer-analysis algorithm that determines an over-approximation of the set of numeric values and addresses that each register and memory location holds at each program point [41]. A *memory region* is an abstract quantity that represents all runtime activation records of a procedure. To represent a set of numeric values and addresses, VSA uses *value-set*s, where a value-set is a map from memory regions to strided intervals. A strided interval consists of a lower bound lb, a stride s, and an upper bound $lb + ks$, and represents the set of numbers $\{lb,\ lb + s,\ lb + 2s,\ ...,\ lb + ks\}$ [160].

The Interpretation of Basetypes and Basetype-Operators. The abstract domain for the integer basetypes is a value-set. The abstract domain for BOOL is Bool3 ($\{$FALSE, MAYBE, TRUE$\}$), where MAYBE means "may be FALSE or may be TRUE". The operators on these domains are described in detail in [160].

The Interpretation of Map-Basetypes and Access/Update Functions. The abstract domain for memory maps (MEMMAP32_8, MEMMAP64_8, etc.) is a dictionary that maps each abstract memory location (i.e., the abstraction of INT32) to a value-set. The abstract domain for register maps (REGMAP32, REGMAP64, etc.) is a dictionary that maps each variable (reg32, reg64, etc.) to a value-set. The abstract domain for flag maps (FLAGMAP) is a dictionary that maps a flag to a Bool3. The *access/update* functions access or update these dictionaries.

VSA uses this transformer evaluator to create an output abstract state, given an instruction and an input abstract state. For example, row 1 of Tab. 3.2 shows the generated VSA transformer for the instruction "add ebx, eax". The VSA evaluator returns a new abstract state in which ebx is updated with the sum of the values of ebx and eax from the input abstract state and the flags are updated appropriately.

3.3.2 Creation of a TC Transformer Generator for ARA

An affine relation is a linear-equality constraint between integer-valued variables. ARA finds affine relations that hold in the program, for a given set of variables. This analysis is used to find

induction-variable relationships between registers and memory locations; these help in increasing the precision of VSA when interpreting conditional branches (§3.2.2.1) [38].

The principle that is used to create a TC transformer generator is as follows: by interpreting the TSL expression that defines the semantics of an individual instruction using an abstract domain in which values represent transformers, each call to interpInstr will residuate a transformer for the instruction. In the case of ARA, the CIR is instantiated so that for each instruction, the generated transformer operates on an abstract domain whose values are sets of matrices that represent affine transformations on registers and memory locations of the state [141].

Interpretation of Basetypes and Basetype-Operators. The abstract domain for the integer basetypes is a set of linear expressions in which variables are either a register or an abstract memory location—the actual representation of the domain is a set of *column*s that consist of an integer constant and an integer coefficient for each variable. This column represents an affine expression over the values that the variables hold at the beginning of the instruction. The basetype operations are defined so that only a set of linear expressions can be generated; any operation that leads to a nonlinear expression, such as Times(eax, ebx), returns TOP, which means that no affine relationship is known to hold.

Interpretation of Map-Basetypes and Access/Update Functions. The abstract domain of the maps for ARA is a set of matrices of size $(N + 1) \times (N + 1)$, where N is the number of variables. This abstraction, which is able to find all affine relationships in an affine program, was defined by Müller-Olm and Seidl [141]. Each *access* function extracts a set of columns associated with the variable it takes as an argument, from the set of matrices for its map argument. Each *update* function creates a new set of matrices that reflects the affine transformation associated with the update to the variable in question.

For each instruction, the ARA transformer relates linear-equality relationships that hold before the instruction to those that hold after execution of the instruction.

3.3.3 Def-Use Analysis (DUA)

Def-Use analysis finds the relationships between *definitions* (*defs*) and *uses* of state components (registers, flags, and memory-locations) for each instruction.

The Interpretation of Basetypes and Basetype-Operators. The abstract domain for the basetypes is a set of *uses* (i.e., abstractions of the map-keys in states, such as registers, flags, and abstract memory locations), and the operators on this domain perform a set union of their arguments' sets.

The Interpretation of Map-Basetypes and Access/Update Functions. The abstract domains of the maps for DUA are dictionaries that map each *def* to a set of *uses*. Each *access* function returns the set of *uses* associated with the key parameter. Each *update* function $update(D, k, S)$, where D is a dictionary, k is one of the state components, and S is a set of *uses*, returns an updated dictionary $D[k \mapsto (D(k) \cup S)]$ (or $D[k \mapsto S]$ if a strong update is sound).

The DUA results (e.g., row 2 of Tab. 3.2) are used to create transformers for several additional analyses, such as GMOD analysis [72], which is an analysis to find modified variables for each function f (including variables modified by functions transitively called from f) and live-flag analysis, which is used in our version of VSA to perform trace-splitting/collapsing (see §3.3.5).

3.3.4 Creation of a UB Transformer Generator for ASI

ASI is a unification-based, flow-insensitive algorithm to identify the structure of aggregates in a program [42]. For each instruction, the transformer generator generates a set of ASI commands, each of which is either a command to *split* a memory region or a command to *unify* some portions of memory (and/or some registers). At analysis time, a client analyzer typically applies the transformer generator to each of the instructions in the program, and then feeds the resulting set of ASI commands to an ASI solver to refine the memory regions.

The Interpretation of Basetypes and Basetype-Operators. The abstract domain for the basetypes is a set of *dataref*s, where a *dataref* is an access on specific bytes of a register or memory. The arithmetic, logical, and bit-vector operations tag *dataref*s as *non-unifiable dataref*s, which means that they will only be used to generate *split*s.

The Interpretation of Map-Basetypes and Access/Update Functions. The abstract domain of the maps for ASI is a set of *split*s and *unification*s. The *access* functions generate a set of *dataref*s associated with a memory location or register. The *update* functions create a set of *unification*s or *split*s according to the *dataref*s of the data argument.

For example, for the instruction "mov [ebx],eax", when ebx holds the abstract address $AR\_foo - 12$, where *AR_foo* is the memory region for the activation records of procedure *foo*, the ASI transformer generator emits one ASI *unification* command "*AR_foo*[-12:-9] :=: eax[0:3]".

3.3.5 Quantifier-Free Bit-Vector (QFBV) Semantics

QFBV semantics provides a way to obtain a symbolic representation—as a formula in first-order quantifier-free bit-vector logic—of an instruction's semantics.

The Interpretation of Basetypes and Basetype-Operators. The abstract domain for the integer basetypes is a set of terms, and each operator constructs a term that represents the operation. The abstract domain for BOOL is a formula, and each BOOL-valued operator constructs a formula that represents the operation.

The Interpretation of Map-Basetypes and Access/Update Functions. The abstract domain for the state components is a dictionary that maps a storage component to a term (or a formula in the case of FLAGMAP). The *access/update* functions retrieve from and update the dictionaries, respectively.

QFBV semantics is useful for a variety of purposes. One use is as auxiliary information in an abstract interpreter, such as the VSA analysis engine, to provide more precise abstract interpretation of branches in low-level code. The issue is that many instruction sets provide separate instructions for (i) setting flags (based on some condition that is tested) and (ii) branching according to the values held by flags.

To address this problem, we use a *trace-splitting/collapsing* scheme [136]. The VSA analysis engine partitions the state at each flag-setting instruction based on live-flag information (which is obtained from an analysis that uses the DUA transformers); a semantic reduction [74] is performed on the split VSA states with respect to a formula obtained from the transformer generated by the

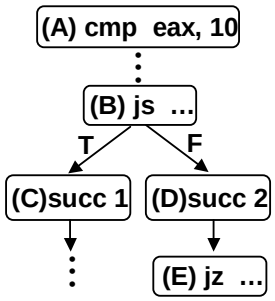


Figure 3.14 An example for trace-splitting

QFBV semantics. The set of VSA states that result are propagated to appropriate successors at the branch instruction that uses the flags.

The cmp instruction (A) in Fig. 3.14, which is a flag-setting instruction, has `sf` and `zf` as live flags because those flags are used at the branch instructions js (B) and jz (E): js and jz jump according to `sf` and `zf`, respectively. After interpretation of (A), the state S is split into four states, S_1, S_2, S_3, and S_4, which are reduced with respect to the formulas φ_1: (eax $-$ 10 $<$ 0) associated with `sf`, and φ_2: (eax $-$ 10 == 0) associated with `zf`.

S_1 := S[`sf`$\mapsto$T] [`zf` $\mapsto$ T] [eax $\mapsto$ *reduce*(S(eax), $\varphi_1 \wedge \varphi_2$)]

S_2 := S[`sf`$\mapsto$T] [`zf` $\mapsto$ F] [eax $\mapsto$ *reduce*(S(eax), $\varphi_1 \wedge \neg\varphi_2$)]

S_3 := S[`sf`$\mapsto$F] [`zf` $\mapsto$ T] [eax $\mapsto$ *reduce*(S(eax), $\neg\varphi_1 \wedge \varphi_2$)]

S_4 := S[`sf`$\mapsto$F] [`zf` $\mapsto$ F] [eax $\mapsto$ *reduce*(S(eax), $\neg\varphi_1 \wedge \neg\varphi_2$)]

Because $\varphi_1 \wedge \varphi_2$ is not satisfiable, S_1 becomes $\perp$. State S_2 is propagated to the true branch of js (i.e., just before (C)), and S_3 and S_4 to the false branch (i.e., just before (D)). Because no flags are live just before (C), the splitting mechanism maintains just a single state, and thus all states propagated to (C)—here there is just one—are collapsed to a single abstract state. Because `zf` is still live until (E), the states S_3 and S_4 are maintained as separate abstract states at (D).

3.4 Measures of Success

As an example of the kind of leverage that TSL provides, the most recent incarnation of CodeSurfer/x86—a revised version whose analysis components are implemented via TSL—uses eight separate reinterpretations generated from the TSL specification of the IA32 instruction set. We estimate that the task of writing transformers (for the eight analysis phases used in CodeSurfer/x86) consumed about 20 man-months; in contrast, we have invested a total of about 1 man-month to write the C++ code for the set of TSL interpretations that are used to generate the replacement components. To this, one should add 10–20 man-days to write the TSL specification for IA32: the current specification for IA32 consists of 2,834 (non-comment, non-blank) lines of TSL.

Because each analysis is defined at the meta-level (i.e., by providing an interpretation for the collection of TSL primitives), abstract transformers for a given analysis can be created automatically for *each* instruction set that is specified in TSL. For instance, from the PowerPC specification (1,370 non-comment, non-blank lines, which took approximately 4 days to write), we were immediately able to generate PowerPC-specific versions of *all* of the analysis components that had been developed for the IA32 instruction set.

It takes approximately 8 seconds (on an Intel Pentium 4 with a 3.00GHz CPU and 2GB of memory, running Centos 4) for the TSL (cross-)compiler to compile the IA32 specification to C++, followed by approximately 20 minutes wall-clock time (on an Intel Pentium 4 with a 1.73GHz CPU and 1.5GB of memory, running Windows XP) to compile the generated C++.

It is natural to ask how the TSL-generated analyses perform compared to their hand-coded counterparts. Due to the nature of the transformers used in one of the analyses that we implemented (affine-relation analysis (ARA) [141]), it was possible to write an algorithm to compare the TSL-generated ARA transformers with the hand-coded ARA transformers that were incorporated in CodeSurfer/x86. On a corpus of 542 instruction instances that covered various opcodes, addressing modes, and operand sizes, we found that the TSL-generated transformers were equivalent in 324 cases and *more precise* than the hand-coded transformers in the remaining 218 cases (40%). For 87 cases, this was because in rethinking how the ARA abstraction could be encoded using TSL

	hand-coded ARA transformers	TSL-generated ARA transformers
time (sec)	0.032	0.281
total # of memory allocs	4,735	31,234
max # of memory allocs	20	682

Figure 3.15 Time (in seconds) and the total/maximum number of memory allocations for getting TSL-generated ARA transformers and hand-coded transformers.

mechanisms, we discovered an easy way to extend [141] to retain some information for 8-, 16-, and 64-bit operations. (In principle, these could have been incorporated into the hand-coded version, too.)

The other 131 cases of improvement can be ascribed to "fatigue factor" on the part of the human programmer: the hand-coded versions adopted a pessimistic view and just treated certain instructions as always assigning an unknown value to the registers that they affected, regardless of the values of the arguments. Because the TSL-generated transformers are based on the ARA interpretation's definitions of the TSL basetype-operators, the TSL-generated transformers were more thorough: a basetype-operator's definition in an interpretation is used in *all* places that the operator arises in the specification of the instruction set's concrete semantics.

We measured time and memory consumption to answer the question "how costly is it to use the TSL-generated analyses". Fig. 3.15 compares the time (in seconds) and memory consumption (in number of memory allocations for matrices, which are used in the representation of abstract elements in the abstract domain for ARA) taken for obtaining 542 TSL-generated ARA transformers with the time and memory for obtaining the corresponding ARA transformers by the hand-coded method that was used in the original CodeSurfer/x86. The TSL-based method takes about 8 times longer than the hand-coded approach and causes about 7 times more memory allocations. In TSL, all the abstract operations (matrix manipulations) are performed at the meta-level essentially in a side-effect-free functional environment. Therefore, there can be many unnecessary memory allocations and object copies at the meta-operator boundaries. However, there is a room for improvement by optimization. Also, TSL still takes less than a second for obtaining 542 ARA transformers. In

the light of the performance measurement of ARA, which is the most memory-intensive analysis we have created using the TSL system, TSL-generated analysis does not cause a significant performance degradation.

We also carried out a study using an algorithm for obtaining "best transformer". For a given instruction I, the TSL QFBV reinterpretation was used to obtain a formula φ_I that expresses the semantics of I. The formula φ_I was then used to obtain (a close approximation to) the best ARA transformer that over-approximates φ_I, using the techniques described in [116, 161]. About 8.5% of the ARA transformers generated via the best-transformer algorithm were more precise than the ARA transformers generated via the TSL-based method. However, there is a trade-off between precision and speed: the best-transformer method is about 600 times slower (as of May 3, 2011) than the TSL-based method.

Leverage

The TSL system provides two dimensions of parameterizability: different instruction sets and different analyses. Each instruction-set specification developer writes an instruction-set semantics, and each analysis developer defines an abstract domain for a desired analysis by giving an interpretation (i.e., the implementations of TSL basetypes, basetype-operators, and *access/update* functions). Given the inputs from these two classes of users, the TSL system automatically generates an analysis component. Note that the work that an analysis developer performs is TSL-specific but *independent* of each language to be analyzed; from the interpretation that defines an analysis, the abstract transformers for that analysis can be generated automatically for *every* instruction set for which one has a TSL specification. Thus, to create $M \times N$ analysis components, the TSL system only requires M specifications of the concrete semantics of instruction sets, and N analysis implementations (Fig. 3.1), i.e., $M + N$ inputs to obtain $M \times N$ analysis-component implementations.

The TSL system provides considerable leverage for implementing analysis tools and experimenting with new ones. New analyses are easily implemented because a clean interface is provided for defining an interpretation.

TSL as a Tool Generator. A tool generator (or tool-component generator) such as YACC [107] takes a declarative description of some desired behavior and automatically generates an implementation of a component that behaves in the desired way. Often the generated component consists of generated tables and code, plus some unchanging *driver* code that is used in each generated tool component. The advantage of a tool generator is that it creates correct-by-construction implementations.

For machine-code analysis, the desired components each consist of a suitable abstract interpretation of the instruction set, together with some kind of analysis driver (a solver for finding the fixed-point of a set of dataflow equations, a symbolic evaluator for performing symbolic execution, etc.). TSL is a system that takes a description of the concrete semantics of an instruction set, a description of an abstract interpretation, and creates an implementation of an abstract interpreter for the given instruction set.

TSL : concrete semantics $\times$ abstract domain $\rightarrow$ abstract semantics.

In that sense, TSL is a tool generator that, for a fixed instruction-set semantics, automatically creates different abstract interpreters for the instruction set.

The reinterpretation mechanism allows TSL to be used to implement *tool-component generators* and *tool generators*. Each implementation of an analysis component's driver (e.g., fixed-point-finding solver, symbolic executor) serves as the unchanging driver for use in different instantiations of the analysis component for different instruction sets. The TSL language becomes the specification language for retargeting that analysis component for different instruction sets:

analyzer generator = abstract-semantics generator + analysis driver.

For tools like CodeSurfer/x86, which incorporates multiple analysis components, we thereby obtain YACC-like tool generators for such tools:

concrete semantics of L $\rightarrow$ Tool/L.

Consistency. In addition to leverage and thoroughness, for a system like CodeSurfer/x86—which uses multiple analysis phases—automating the process of creating abstract transformers

ensures *semantic consistency*; that is, because analysis implementations are generated from a *single* specification of the instruction set's concrete semantics, this guarantees that a *consistent* view of the concrete semantics is adopted by all of the analyses used in the system.

3.5 Related Work

In this section, we discuss work from various domains that relates to TSL. §3.5.1 compares the way we use the technique of reinterpreting TSL's base-types and meta-operators to the concept of *refactoring* as in the original work on semantic reinterpretation [110, 134, 144, 148]. §3.5.2 discusses some instruction-set-description languages developed for various purposes. §3.5.3 presents various existing systems for creating analyzers and transformers.

3.5.1 Semantic Reinterpretation

As discussed in §3.1, *semantic reinterpretation* involves refactoring the specification of a language's concrete semantics into a suitable form by introducing appropriate *combinators* that are subsequently redefined to create the different subject-language interpretations.

Semantic Reinterpretation Versus Standard Abstract Interpretation. Semantic reinterpretation [110, 134, 144, 148] is a form of abstract interpretation [73], but differs from the way abstract interpretation is normally applied: in standard abstract interpretation, one reinterprets the constructs of each *subject language*; in contrast, with semantic reinterpretation one reinterprets the constructs of the *meta-language*. Standard abstract interpretation helps in creating semantically sound *tools*; semantic reinterpretation helps in creating semantically sound *tool generators*. In particular, if you have N subject languages and M analyses, with semantic reinterpretation you obtain $N \times M$ analyzers by writing just $N + M$ specifications: concrete semantics for N subject languages and M reinterpretations. With the standard approach, one must write $N \times M$ abstract semantics.

As originally proposed, semantic reinterpretation permits arbitrary refactoring of a semantic specification so that the desired outcome can be achieved via reinterpretation of any combinators introduced. In contrast, in TSL—although it is possible to introduce combinators and refactor them—the primary mechanism is to reinterpret the base-types and meta-operators of the TSL meta-language. This approach is particularly convenient for a system to generate *multiple* analysis components from a single specification of a language's concrete semantics.

Semantic Reinterpretation Versus Translation to a Common Intermediate Representation. The mapping of subject-language constructs to meta-language operations that one defines as part of the semantic-reinterpretation approach resembles a translation to a common intermediate representation (CIR) *data structure*. Thus, another approach to obtaining "systematic" reinterpretations that are similar to semantic reinterpretations—in that they apply to multiple subject languages—would be to translate subject-language programs to a CIR, and then create various interpreters that implement different abstract interpretations of the node types of the CIR data structure. Each interpreter would then be applied to (the translation of) programs in any subject language L for which one has defined an L-to-CIR translator. Compared with interpreting objects of a CIR data type, the advantages of semantic reinterpretation (i.e., reinterpreting the constructs of the *meta-language*) are

1. The presentation of our ideas is simpler because one does not have to introduce an additional language of trees for representing CIR objects.
2. With semantic reinterpretation, there is no explicit CIR data structure to be interpreted. In essence, semantic reinterpretation removes a level of interpretation, and hence generated analyzers should run faster.

Micro-semantics and Macro-semantics

Pleban and Lee proposed the MESS system, a prototype implementation of a compiler generator, which is based on a semantic-definition style, called *high-level semantics* [154]. The high-level semantics was designed to overcome fundamental problems that have precluded the generation of

realistic compilers from traditional denotational specifications. They introduced a separation of the semantic definition of a programming language into two distinct specifications, called *macro-semantics* and *micro-semantics*. The macro-semantics of a language is defined by a collection of semantic functions that map syntactic phrases, compositionally, to terms of a semantic algebra. The micro-semantics specifies the meaning of a semantic algebra.

3.5.2 Instruction-Set-Description Languages

There have been many specification languages for instruction sets and many purposes to which they have been applied. Some were designed for hardware simulation, such as cycle simulation and pipeline simulation [152, 137]. Others have been used to generate an emulator for compiler-optimization testing [113, 77]. TDL [113] is a hardware-description language that supports the retargeting of back-end phases, such as analyses and optimizations relevant to instruction scheduling, register assignment, and functional-unit binding. The New Jersey machine-code toolkit [158] addresses concrete syntactic issues (instruction decoding, instruction encoding, etc.). While some of the existing *languages* would have been satisfactory for our purposes, their *runtime components* were not satisfactory, which necessitated creating our own implementation.

In our work, we needed a mechanism to create abstract interpreters of instruction-set specifications. There are (at least) four issues that arise: during the abstract interpretation of each transformer, the abstract interpreter must be able to

- execute over abstract states,
- execute both branches of a conditional expression,
- compare abstract states and terminate abstract execution when a fixed point is reached, and
- apply widening operators, if necessary, to ensure termination.

As far as we know, TSL is the first system with an instruction-set-specification language and support for such mechanisms.

Although this chapter only discusses the application of TSL to low-level instruction sets, we believe that only small extensions would be needed to be able to apply TSL to source-code languages (i.e., to create language-independent analyzers for source-level IRs), as well as bytecode.

The main obstacle is that the concrete semantics of a source-code language generally uses an execution state based on a stack of variable-to-value (or variable-to-location, location-to-value) maps. For a low-level language, the state incorporates an address-based memory model, for which the TSL language provides appropriate primitives.

Functional languages as instruction-set-description language. Harcourt et al. used ML to specify the semantics of instruction sets [99]. LISAS [71] is an instruction-set-description language that was subsequently developed based on their experience using ML. Those two approaches particularly influenced the design of the TSL language.

λ-RTL. TSL shares some of the same goals as λ-RTL [157] (i.e., the ability to specify the semantics of an instruction set and to support multiple clients that make use of a single specification). The two languages were both influenced by ML, but different choices were made about what aspects of ML to retain: λ-RTL is higher-order, but without datatype constructors and recursion; TSL is first-order, but supports both datatype constructors and recursion. As discussed in §3.2.2.2, recursion is not often used in specifications, but is needed for handling some loop-iteration instructions, such as the IA32 string-manipulation instructions and the PowerPC multiple-word load/store instructions. The choices made in the design and implementation of TSL were driven by the goal of being able to define multiple abstract interpretations of an instruction-sets semantics.

Insruction-Set Processor Specifications (ISPS). Siewiorek et al. [170] proposed an operational hardware specification language, the ISP (Instruction-Set Processor) notation, for describing the instructions in a processor and how they are implemented, aiming to automate the generation of software, the evaluation of computer architectures, and the certification of implementations.

They divide a computer system into several levels including the *program level*, which the ISP notation is designed to properly describe. Their design of the ISP notation is based on two principles: (i) the effect of each instruction can be expressed entirely in terms of the information held in the current memory (state); the components of the program level are a set of memories and a set of operations. The ISP notation is designed for specifying that a given operation of a processor is

performed on a specific data structure that the set of memories hold, and (ii) all the data operations can be characterized as working on various *data-types*; each data-type requires distinct operations to process the values of a data-type. A processor can be completely described at the ISP level by giving its *instruction set* and its *interpreter* in terms of its *operations*, *data-types*, and *memories*. TSL relies on the same principles.

3.5.3 Systems for Generating Analyzers

Some systems for representing and analyzing programs are (mainly) targeted for a single language. For instance, SOOT [23] is a powerful and flexible analysis/optimization framework that supports analysis and transformation of Java bytecode. One method to support the retargeting of analyses to different languages is to create a package that supports a family of program analyses that different front ends can use to create analysis components. Examples include BDDBDDB [184], Banshee [117], the Parma Polyhedra Library [21], WPDS++ [115], and WALi [114]. The writer of each client front end needs to encode the semantics of his language by creating appropriate transformers for each statement and condition in the language's IR, using the package's API (or input language).

WALA [30] supports a common intermediate form (Common Abstract Syntax Tree), from which multiple additional IRs (e.g., CFGs and SSA-form) can be generated, and multiple analyses can be performed that use these IRs. Thus, this is similar to the package approach, but supports a multiplicity of analyses.

In contrast to the package approach, TSL provides a domain-specific language for specifying the semantics of instruction sets. With this approach, the ISS developer concentrates on specifying the concrete operational semantics of his language, using TSL, and a multiplicity of analyzers are then created automatically. Analysis developers can incorporate different analysis packages into the TSL framework by implementing appropriate abstract operations that over-approximate the semantics of a fixed set of TSL operations (that have a well-defined semantics). (Any of the aforementioned packages could be used for creating TSL-based analyses; currently, WALi is used for all of the TC-style analyzers that have been developed for use with TSL so far.)

There are two analysis systems, TVLA [28] and the optimizer flow-function inference system developed by Rice et al. [164], in which sound analysis transformers are generated automatically from a concrete operational semantics, plus a specification of an abstraction (either via the abstraction function (TVLA) or the concretization function (Rice et al.)). In our system, we rely on the analysis developer to supply sound abstract operations. While this places an additional burden on developers, once an analysis is developed it can be used with each instruction set specified in TSL. Moreover,

- The analyses that we support are much more efficient than those that can be created with TVLA and apply to our intended domain of application (low-level code).
- Some of the analyses that we use, such as ARA [141], appear to be beyond the power of the heuristics-based transformer-generation methods developed by Rice et al.

Chapter 4

Symbolic Analysis via Semantic Reinterpretation

The use of symbolic-reasoning primitives for *forward symbolic evaluation*, *weakest liberal precondition* ($\mathcal{WLP}$), and *symbolic composition* has experienced a resurgence in program-analysis tools because of the power that they provide when exploring a program's state space.

Model-checking tools, such as SLAM [46] and BLAST [102], as well as hybrid concrete/symbolic program-exploration tools, such as DART [94], CUTE [167], YOGI [98], SAGE [95], BITSCOPE [54], and DASH [49] use forward symbolic evaluation, $\mathcal{WLP}$, or both. An important subroutine in these tools is to determine the following: given a path π in the program, is π feasible (i.e., executable)?

Given path π, symbolic evaluation is used to construct a path formula ψ for π such that π is feasible if and only if ψ is satisfiable. Moreover, a model of ψ can be used to create an input for the program that causes execution to follow path π.

Symbolic evaluation is used to create path formulas. To determine whether a path π is executable, an SMT solver is used to determine whether π's path formula is satisfiable, and if so, to generate an input that drives the program down π. Some of the aforementioned tools also use $\mathcal{WLP}$ to identify new predicates that split part of a program's state space [46, 49]. Proof-carrying code systems [145] use $\mathcal{WLP}$ to create verification conditions.

Bug-finding tools, such as ARCHER [187] and SATURN [186], as well as commercial bug-finding products, such as Coverity's PREVENT [7] and GrammaTech's CODESONAR [4], use symbolic composition. Formulas are used to summarize a portion of the behavior of a procedure. Suppose that procedure P calls Q at call-site c, and that r is the site in P to which control returns after the call at c. When c is encountered during the exploration of P, such tools perform the

symbolic composition of the formula that expresses the behavior along the path $[\mathit{entry}_P, \ldots, c]$ explored in P with the formula that captures the behavior of Q to obtain a formula that expresses the behavior along the path $[\mathit{entry}_P, \ldots, r]$.

Motivation. The standard approach to implementing each of the symbolic-analysis primitives for a programming language of interest (which we call the subject language) is to create hand-written *translation procedures*—one per symbolic-analysis primitive—that convert subject-language commands into appropriate formulas. Such an approach can be extremely tedious. It is also error prone: a system can contain subtle inconsistency bugs if the different translation procedures adopt different "views" of the semantics.

One manifestation of an inconsistency bug would be that if one performs symbolic evaluation of a path π starting from a state that satisfies $\psi = \mathcal{WLP}(\pi, \varphi)$, the resulting symbolic state does not entail φ. Such bugs undermine the soundness of an analysis tool.

The consistency problem is compounded by the issue of aliasing: most subject languages permit memory states to have complicated aliasing patterns, but usually it is not obvious that aliasing is treated consistently across implementations of symbolic evaluation, $\mathcal{WLP}$, and symbolic composition.

Such bugs are easy to introduce because each translation procedure must encode the subject language's semantics; however, the encodings for symbolic evaluation, $\mathcal{WLP}$, and symbolic composition have different flavors.

Our own interest is in analyzing machine code, such as x86 and PowerPC. Unfortunately, as discussed in §2.4, machine-code instruction sets have hundreds of instructions, as well as other complicating factors, such as the use of separate instructions to set flags (based on the condition that is tested) and to branch according to the flag values, the ability to perform address arithmetic and dereference computed addresses (hence memory states can have complicated aliasing patterns), non-aligned memory accesses, etc. To appreciate the need for tool support for creating symbolic-analysis primitives for real machine-code languages, consult the Intel instruction-set reference manual ([31, §3.2] and [32, §4.1]), and imagine writing three separate encodings of

each instruction's semantics to implement symbolic evaluation, $\mathcal{WLP}$, and symbolic composition. Some tools (e.g., [54, 95]) need an instruction-set emulator, in which case a fourth encoding of the semantics is also required.

Our approach. To address these issues, this chapter presents a way to automatically obtain mutually-consistent, correct-by-construction implementations of symbolic primitives, by *generating* them from a specification of the subject language's concrete semantics.

The semantics of the basic symbolic-reasoning primitives are easy to state; for instance, if $\tau(\sigma, \sigma')$ is a 2-state formula that represents the semantics of an instruction, then $\mathcal{WLP}(\tau, \varphi)$ can be expressed as $\forall\sigma'.(\tau(\sigma, \sigma') \Rightarrow \varphi(\sigma'))$. However, this formula uses quantification over states—i.e., *second-order quantification*—whereas SMT solvers, such as Yices [82] and Z3 [78], support only *quantifier-free first-order* logic. Hence, such a formula cannot be used directly.

For a simple language that has only `int`-valued variables, it is easy to recast matters in first-order logic. For instance, the $\mathcal{WLP}$ of postcondition φ with respect to an assignment statement $var = rhs$; can be obtained by substituting *rhs* for all (free) occurrences of *var* in φ: $\varphi[var \leftarrow rhs]$. For real-world programming languages, however, the situation is more complicated. For instance, for languages with pointers, Morris's rule of substitution [138] requires taking into account all possible aliasing combinations. In general, tool builders need to create implementations of symbolic primitives for full languages, and hence must be prepared to accommodate whatever features the language supports.

We present a method to obtain quantifier-free, first-order-logic formulas for (a) symbolic evaluation of a single command, (b) $\mathcal{WLP}$ with respect to a single command, and (c) symbolic composition for a class of formulas that express state transformations. The generated implementations are guaranteed to be mutually consistent, and also to be consistent with an instruction-set emulator (for concrete execution) that is generated from the same specification of the subject language's concrete semantics.

Primitives (a) and (b) immediately extend to compound operations over a given program path for use in forward and backwards symbolic evaluation, respectively; see §4.5. (The design of client

algorithms that use such primitives to perform state-space exploration is an orthogonal issue that is outside the scope of this chapter.)

Achievements and Contributions. We used the approach described in the paper to create a "YACC-like" tool for generating mutually-consistent, correct-by-construction implementations of symbolic-analysis primitives for instruction sets (§4.7). The input is a specification of an instruction set's concrete semantics; the output is a triple of C++ functions that implement the three symbolic-analysis primitives—(1) translation of an instruction into a formula, (2) $\mathcal{WLP}$ with respect to an instruction, and (3) symbolic composition. The tool has been used to generate such primitives for x86 and PowerPC. To accomplish this, we leveraged TSL, as the implementation platform for defining the necessary reinterpretations.

The contributions of the work described in this chapter lie in the insights that went into defining the specific reinterpretations that we use to obtain mutually-consistent, correct-by-construction implementations of the symbolic-analysis primitives, and the discovery that $\mathcal{WLP}$ could be obtained by using two different reinterpretations working in tandem. The chapter's other contributions are summarized as follows:

- We present a new application for semantic reinterpretation (§4.1), namely, to create implementations of the basic primitives for symbolic reasoning (§4.3 and 4.4). In particular, two key insights allowed us to obtain the primitives for $\mathcal{WLP}$ and symbolic composition:
  - The first insight was that we could apply semantic reinterpretation in a new context, namely, to the interpretation function of a *logic* (§4.3).
  - The second insight was to define a particular form of state-transformation formula—called a structure-update expression (see §4.2.1)—to be a first-class notion in the logic, which allows such formulas (i) to serve as a replacement domain in various reinterpretations, and (ii) to be reinterpreted themselves (§4.3).
- We show how reinterpretation can automatically create a $\mathcal{WLP}$ primitive that implements Morris's rule of substitution for a language with pointers [138] (§4.3).

- We conducted an experiment that used the generated symbolic-evaluation primitive on real x86 code. The experiment showed that using an exact symbolic-evaluation primitive, as opposed to one that approximates the real semantics, is slower by a factor of 1.07 but is dramatically more accurate (§4.7).

Moreover, we demonstrate that this approach to creating symbolic-analysis primitives can handle languages with pointers and address arithmetic (§4.3 and 4.4). For expository purposes, simplified languages are used throughout. Our discussion of machine code (§4.2.3 and 4.4) is based on a greatly simplified fragment of the x86 instruction set; however, our implementation (§4.7) works on code from real x86 programs compiled from C++ source code, including C++ STL, using Visual Studio.

Organization. The remainder of this chapter is organized as follows: §4.2 defines the logic that we use, as well a simple source-code language (PL) and an idealized machine-code language (MC). §4.3 discusses how to use reinterpretation to obtain the three symbolic-analysis primitives for PL. §4.4 addresses reinterpretation for MC. §4.5 explains how other language constructs beyond those found in PL and MC can be handled. §4.6 describes how non-determinism can be incorporated into our approach. §4.7 describes how we used the TSL system for the implementation, and also presents the experiment carried out with the implementation. §4.8 discusses related work. §4.9 presents some conclusions. Correctness proofs can be found in Appendix B.

4.1 Semantic Reinterpretation

This section presents the basic principles of semantic reinterpretation in the context of abstract interpretation. We use a simple language of assignments, and define the concrete semantics and an abstract sign-analysis semantics via semantic reinterpretation.

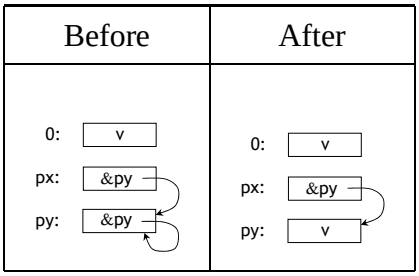
Example 4.1 [Adapted from [134].] Consider the following fragment of a denotational semantics, which defines the meaning of assignment statements over variables that hold signed 32-bit `int`

(a)

s_1: $x = x \oplus y;$
s_2: $y = x \oplus y;$
s_3: $x = x \oplus y;$

(b)

t_1: $*px = *px \oplus *py;$
t_2: $*py = *px \oplus *py;$
t_3: $*px = *px \oplus *py;$



(c)

```
[1] mov eax, [ebp−10]
[2] xor eax, [ebp−14]
[3] mov [ebp−10], eax
[4] mov eax, [ebp−10]
[5] xor eax, [ebp−14]
[6] mov [ebp−14], eax
[7] mov eax, [ebp−10]
[8] xor eax, [ebp−14]
[9] mov [ebp−10], eax
```

(d)

Figure 4.1 (a) Code fragment that swaps two `int`s; (b) code fragment that swaps two `int`s using pointers; (c) possible before and after configurations for code fragment (b): the swap is unsuccessful due to aliasing; (d) x86 machine code (in Intel syntax) corresponding to (a).

values (where $\oplus$ denotes exclusive-or):

$$I \in \textit{Id} \qquad E \in \textit{Expr} ::= I \mid E_1 \oplus E_2 \mid \ldots$$

$$S \in \textit{Stmt} ::= I = E; \qquad \sigma \in \textit{State} = \textit{Id} \rightarrow \textit{Int32}$$

$$\mathcal{E} : \textit{Expr} \rightarrow \textit{State} \rightarrow \textit{Int32}$$

$$\mathcal{E}[\![I]\!]\sigma = \sigma I$$

$$\mathcal{E}[\![E_1 \oplus E_2]\!]\sigma = \mathcal{E}[\![E_1]\!]\sigma \oplus \mathcal{E}[\![E_2]\!]\sigma$$

$$\mathcal{I} : \textit{Stmt} \rightarrow \textit{State} \rightarrow \textit{State}$$

$$\mathcal{I}[\![I = E;]\!]\sigma = \sigma[I \mapsto \mathcal{E}[\![E]\!]\sigma]$$

We use the notation "$\sigma[I \mapsto v]$," to mean the *State* that acts like σ except that argument I is mapped to v. The function $\mathcal{I}$ can be understood as an *interpreter* for the language: $(\mathcal{I}[\![s]\!]\sigma)$ is the state that results from executing statement s on the state σ. A sequence of statements can be executed by repeatedly calling $\mathcal{I}$. For instance, consider the program shown in Fig. 4.1(a), which swaps two `int`s. Execution of this code, starting from the state $\sigma_0 = \{x \mapsto -1, y \mapsto 2\}$ can be achieved as follows:

$$\begin{array}{lcll}
\sigma_0 & := & \{x \mapsto -1, y \mapsto 2\} & \\
\sigma_1 & := & \mathcal{I}[\![s_1 : x = x \oplus y;]\!]\sigma_0 & = \{x \mapsto -3, y \mapsto 2\} \\
\sigma_2 & := & \mathcal{I}[\![s_2 : y = x \oplus y;]\!]\sigma_1 & = \{x \mapsto -3, y \mapsto -1\} \\
\sigma_3 & := & \mathcal{I}[\![s_3 : x = x \oplus y;]\!]\sigma_2 & = \{x \mapsto 2, y \mapsto -1\}
\end{array}$$

The languages derivable from *Expr* and *State* define the subject language. The semantics is defined using a *meta-language*. In this example, the meta-language has one base type (*Int32*). It supports defining map types (*State* $=$ *Id* $\rightarrow$ *Int32*) and user-defined functions ($\mathcal{E}$ and $\mathcal{I}$). It also supports operations on base-type values (e.g., "$\_ \oplus \_$"), map-access operations (σI), map-update operations ($\sigma[I \mapsto \mathcal{E}[\![E]\!]\sigma]$), and invocation of user-defined functions ($\mathcal{E}[\![E]\!]\sigma$).

To highlight better the role of the meta-language, we introduce names for certain aspects of the meta-language. For instance, the one base type, whose standard interpretation is *Int32*, will be called *Val*. We also introduce names for the following operators:

- "$\_$ *xor* $\_$", whose standard interpretation is "$\_ \oplus \_$".
- *lookup*, for map-access operations.
- *store*, for map-update operations.

The specification given earlier is thus rewritten as follows:

$$\begin{array}{l}
\mathit{xor} : \mathit{Val} \rightarrow \mathit{Val} \rightarrow \mathit{Val} \\
\mathit{lookup} : \mathit{State} \rightarrow \mathit{Id} \rightarrow \mathit{Val} \\
\mathit{store} : \mathit{State} \rightarrow \mathit{Id} \rightarrow \mathit{Val} \rightarrow \mathit{State} \\
\mathcal{E} : \mathit{Expr} \rightarrow \mathit{State} \rightarrow \mathit{Val} \\
\mathcal{E}[\![I]\!]\sigma = \mathit{lookup}\ \sigma\ I \\
\mathcal{E}[\![E_1 \oplus E_2]\!]\sigma = \mathcal{E}[\![E_1]\!]\sigma\ \mathit{xor}\ \mathcal{E}[\![E_2]\!]\sigma \\
\mathcal{I} : \mathit{Stmt} \rightarrow \mathit{State} \rightarrow \mathit{State} \\
\mathcal{I}[\![I = E;]\!]\sigma = \mathit{store}\ \sigma\ I\ \mathcal{E}[\![E]\!]\sigma
\end{array}$$

For the concrete (or "standard") semantics, the meta-language types and operators are defined as follows:

$$v \in \mathit{Val}_{\mathit{std}} = \mathit{Int32} \qquad \mathit{lookup}_{\mathit{std}} = \lambda\sigma.\lambda I.\sigma I$$
$$\mathit{State}_{\mathit{std}} = \mathit{Id} \to \mathit{Val} \qquad \mathit{store}_{\mathit{std}} = \lambda\sigma.\lambda I.\lambda v.\sigma[I \mapsto v]$$
$$\mathit{xor}_{\mathit{std}} = \lambda v_1.\lambda v_2.v_1 \oplus v_2$$

Different abstract interpretations of the same language can be defined by using the same semantic specification, but by giving different interpretations of the base types, function types, and operators of the meta-language. For example, for sign analysis, assuming that *Int32* values are represented in two's complement, the meta-language is reinterpreted as follows:[1]

$$\begin{aligned} v \in \mathit{Val}_{\mathit{abs}} &= \{\mathit{neg}, \mathit{zero}, \mathit{pos}, \top\} \\ \mathit{State}_{\mathit{abs}} &= \mathit{Id} \to \mathit{Val}_{\mathit{abs}} \\ \mathit{lookup}_{\mathit{abs}} &= \lambda\sigma.\lambda I.\sigma I \\ \mathit{store}_{\mathit{abs}} &= \lambda\sigma.\lambda I.\lambda v.\sigma[I \mapsto v] \end{aligned}$$

$\mathit{xor}_{\mathit{abs}} = \lambda v_1.\lambda v_2.$

		v_2			
		neg	*zero*	*pos*	$\top$
v_1	*neg*	$\top$	*neg*	*neg*	$\top$
	zero	*neg*	*zero*	*pos*	$\top$
	pos	*neg*	*pos*	$\top$	$\top$
	$\top$	$\top$	$\top$	$\top$	$\top$

Essentially, this redefines (or abstracts) the set of values $\mathit{Val}_{\mathit{std}}$ to $\mathit{Val}_{\mathit{abs}}$ and redefines the operators (like *xor*) to operate on the abstract values.

For the code fragment shown in Fig. 4.1(a), sign-analysis reinterpretation creates abstract transformers that, given the initial abstract state $\sigma_0 = \{x \mapsto \mathit{neg}, y \mapsto \mathit{pos}\}$, produce the abstract states shown in Fig. 4.2. □

[1] For the two's-complement representation, $\mathit{pos}\ \mathit{xor}_{\mathit{abs}}\ \mathit{neg} = \mathit{neg}\ \mathit{xor}_{\mathit{abs}}\ \mathit{pos} = \mathit{neg}$ because, for all combinations of values represented by *pos* and *neg*, the high-order bit of the result is set, which means that the result is always negative. However, $\mathit{pos}\ \mathit{xor}_{\mathit{abs}}\ \mathit{pos} = \mathit{neg}\ \mathit{xor}_{\mathit{abs}}\ \mathit{neg} = \top$ because the concrete result could be either 0 or positive, and $\mathit{zero} \sqcup \mathit{pos} = \top$.

$$\sigma_0 := \{x \mapsto \mathit{neg}, y \mapsto \mathit{pos}\}$$
$$\sigma_1 := \mathcal{I}[\![s_1 : x = x \oplus y;]\!]\sigma_0 = \mathit{store}_{\mathit{abs}}\ \sigma_0\ x\ (\mathit{neg}\ \mathit{xor}_{\mathit{abs}}\ \mathit{pos}) = \{x \mapsto \mathit{neg}, y \mapsto \mathit{pos}\}$$
$$\sigma_2 := \mathcal{I}[\![s_2 : y = x \oplus y;]\!]\sigma_1 = \mathit{store}_{\mathit{abs}}\ \sigma_1\ y\ (\mathit{neg}\ \mathit{xor}_{\mathit{abs}}\ \mathit{pos}) = \{x \mapsto \mathit{neg}, y \mapsto \mathit{neg}\}$$
$$\sigma_3 := \mathcal{I}[\![s_3 : x = x \oplus y;]\!]\sigma_2 = \mathit{store}_{\mathit{abs}}\ \sigma_2\ x\ (\mathit{neg}\ \mathit{xor}_{\mathit{abs}}\ \mathit{neg}) = \{x \mapsto \top, y \mapsto \mathit{neg}\}.$$

Figure 4.2 Application of the abstract transformers created by the sign-analysis reinterpretation to the initial abstract state $\sigma_0 = \{x \mapsto \mathit{neg}, y \mapsto \mathit{pos}\}$.

4.2 A Logic and Two Programming Languages

This section defines quantifier-free first-order bit-vector logic, L, a simple source-code language, PL, which only has `int`-valued variables and pointer variables, and a simple machine-code language MC.

4.2.1 L: A Quantifier-Free Bit-Vector Logic with Finite Functions

The logic L is quantifier-free first-order bit-vector logic over a vocabulary of constant symbols ($I \in \mathit{Id}$) and function symbols ($F \in \mathit{FuncId}$). Strictly speaking, we work with various instantiations of L, denoted by $L[\mathrm{PL}]$ and $L[\mathrm{MC}]$, in which the vocabularies of function symbols are chosen to describe aspects of the values used by, and computations performed by, the programming languages PL and MC, respectively.

We distinguish the syntactic symbols of L from their counterparts in PL (§4.1 and 4.2.2) by using boxes around L's symbols.

$$\begin{aligned}
c \in C_{\mathit{Int32}} &= \{0, 1, \ldots\} \\
\mathit{op2}_L \in \mathit{BinOp}_L &= \{\boxed{+}, \boxed{\text{-}}, \boxed{\oplus}, \ldots\} \\
\mathit{rop}_L \in \mathit{RelOp}_L &= \{\boxed{=}, \boxed{\neq}, \boxed{<}, \boxed{>}, \ldots\} \\
\mathit{bop}_L \in \mathit{BoolOp}_L &= \{\boxed{\&\&}, \boxed{||}, \ldots\}
\end{aligned}$$

The rest of the syntax of $L[\cdot]$ is defined as follows:

$$I \in \mathit{Id}, T \in \mathit{Term}, \varphi \in \mathit{Formula},$$
$$F \in \mathit{FuncId}, \mathit{FE} \in \mathit{FuncExpr}, U \in \mathit{StructUpdate}$$
$$T ::= c \mid I \mid T_1\, \mathit{op2}_L\, T_2 \mid \mathit{ite}(\varphi, T_1, T_2) \mid \mathit{FE}(T)$$
$$\varphi ::= \boxed{\mathbb{T}} \mid \boxed{\mathbb{F}} \mid T_1\, \mathit{rop}_L\, T_2 \mid \boxed{\neg}\varphi_1 \mid \varphi_1\, \mathit{bop}_L\, \varphi_2$$
$$\mathit{FE} ::= F \mid \mathit{FE}_1[T_1 \mapsto T_2]$$
$$U ::= (\{I_i \hookleftarrow T_i\}, \{F_j \hookleftarrow \mathit{FE}_j\})$$

A *Term* of the form $\mathit{ite}(\varphi, T_1, T_2)$ represents an if-then-else expression. Names of the form $F \in \mathit{FuncId}$, possibly with subscripts and/or primes, are function symbols. A *FuncExpr* of the form $\mathit{FE}_1[T_1 \mapsto T_2]$ denotes a *function-update expression*. A *StructUpdate* of the form $(\{I_i \hookleftarrow T_i\}, \{F_j \hookleftarrow \mathit{FE}_j\})$ is called a *structure-update expression*. It specifies a structure-transformation operation that yields a structure in which the identifier I_i is updated to the value of term T_i, and the function identifier F_j is updated to the value of function-expression FE_j. The subscripts i and j implicitly range over certain index sets, which will be omitted to reduce clutter. To emphasize that I_i and F_j refer to next-state quantities, we sometimes write structure-update expressions with primes: $(\{I'_i \hookleftarrow T_i\}, \{F'_j \hookleftarrow \mathit{FE}_j\})$. $\{I'_i \hookleftarrow T_i\}$ specifies the updates to the interpretations of the constant symbols and $\{F'_j \hookleftarrow \mathit{FE}_j\}$ specifies the updates to the interpretations of the function symbols (see below). Thus, a structure-update expression $(\{I'_i \hookleftarrow T_i\}, \{F'_j \hookleftarrow \mathit{FE}_j\})$ can be thought of as a kind of restricted 2-vocabulary (i.e., 2-state) formula $\bigwedge_i (I'_i = T_i) \wedge \bigwedge_j (F'_j = \mathit{FE}_j)$. We define U_{id} to be

$$(\{I' \hookleftarrow I \mid I \in \mathit{Id}\}, \{F' \hookleftarrow F \mid F \in \mathit{FuncId}\}).$$

Semantics of L. The semantics of $L[\cdot]$ is defined in terms of a *logical structure*, which gives meaning to the *Id* and *FuncId* symbols of the logic's vocabulary:

$$\iota \in \mathit{LogicalStruct} = (\mathit{Id} \to \mathit{Val}) \times (\mathit{FuncId} \to (\mathit{Val} \to \mathit{Val})).$$

$(\iota{\uparrow}1)$ assigns meanings to constant symbols, and $(\iota{\uparrow}2)$ assigns meanings to function symbols. ("$(p{\uparrow}1)$" and "$(p{\uparrow}2)$" denote the 1st and 2nd components, respectively, of a pair p.)

$$
\begin{aligned}
\mathit{const} &: C_{\mathit{Int32}} \to \mathit{Val} \\
\mathit{cond}_L &: \mathit{BVal} \to \mathit{Val} \to \mathit{Val} \to \mathit{Val} \\
\mathit{lookupId} &: \mathit{LogicalStruct} \to \mathit{Id} \to \mathit{Val} \\
\mathit{binop}_L &: \mathit{BinOp}_L \to (\mathit{Val} \times \mathit{Val} \to \mathit{Val}) \\
\mathit{relop}_L &: \mathit{RelOp}_L \to (\mathit{Val} \times \mathit{Val} \to \mathit{BVal}) \\
\mathit{boolop}_L &: \mathit{BoolOp}_L \to (\mathit{BVal} \times \mathit{BVal} \to \mathit{BVal}) \\
\mathit{lookupFuncId} &: \mathit{LogicalStruct} \to \mathit{FuncId} \to (\mathit{Val} \to \mathit{Val}) \\
\mathit{access} &: (\mathit{Val} \to \mathit{Val}) \times \mathit{Val}) \to \mathit{Val} \\
\mathit{update} &: ((\mathit{Val} \to \mathit{Val}) \times \mathit{Val} \times \mathit{Val}) \to (\mathit{Val} \to \mathit{Val})
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{T} &: \mathit{Term} \to \mathit{LogicalStruct} \to \mathit{Val} \\
\mathcal{T}[\![c]\!]\iota &= \mathit{const}(c) \\
\mathcal{T}[\![I]\!]\iota &= \mathit{lookupId}\ \iota\ I \\
\mathcal{T}[\![T_1\ \mathit{op2}_L\ T_2]\!]\iota &= \mathcal{T}[\![T_1]\!]\iota\ \mathit{binop}_L(\mathit{op2}_L)\ \mathcal{T}[\![T_2]\!]\iota \\
\mathcal{T}[\![\mathit{ite}(\varphi, T_1, T_2)]\!]\iota &= \mathit{cond}_L(\mathcal{F}[\![\varphi]\!]\iota, \mathcal{T}[\![T_1]\!]\iota, \mathcal{T}[\![T_2]\!]\iota) \\
\mathcal{T}[\![\mathit{FE}(T_1)]\!]\iota &= \mathit{access}(\mathcal{FE}[\![\mathit{FE}]\!]\iota, \mathcal{T}[\![T_1]\!]\iota)
\end{aligned}
\qquad
\begin{aligned}
\mathcal{F} &: \mathit{Formula} \to \mathit{LogicalStruct} \to \mathit{BVal} \\
\mathcal{F}[\![\boxed{\mathbb{T}}]\!]\iota &= \mathbb{T} \\
\mathcal{F}[\![\boxed{\mathbb{F}}]\!]\iota &= \mathbb{F} \\
\mathcal{F}[\![T_1\ \mathit{rop}_L\ T_2]\!]\iota &= \mathcal{T}[\![T_1]\!]\iota\ \mathit{relop}_L(\mathit{rop}_L)\ \mathcal{T}[\![T_2]\!]\iota \\
\mathcal{F}[\![\boxed{\neg}\,\varphi_1]\!]\iota &= \neg\mathcal{F}[\![\varphi_1]\!]\iota \\
\mathcal{F}[\![\varphi_1\ \mathit{bop}_L\ \varphi_2]\!]\iota &= \mathcal{F}[\![\varphi_1]\!]\iota\ \mathit{boolop}_L(\mathit{bop}_L)\ \mathcal{F}[\![\varphi_2]\!]\iota
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{FE} &: \mathit{FuncExpr} \to \mathit{LogicalStruct} \to (\mathit{Val} \to \mathit{Val}) \\
\mathcal{FE}[\![F]\!]\iota &= \mathit{lookupFuncId}\ \iota\ F \\
\mathcal{FE}[\![\mathit{FE}_1[T_1 \mapsto T_2]]\!]\iota &= \mathit{update}(\mathcal{FE}[\![\mathit{FE}_1]\!]\iota, \mathcal{T}[\![T_1]\!]\iota, \mathcal{T}[\![T_2]\!]\iota)
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{U} &: \mathit{StructUpdate} \to \mathit{LogicalStruct} \to \mathit{LogicalStruct} \\
\mathcal{U}[\![(\{I_i \hookleftarrow T_i\}, \{F_j \hookleftarrow \mathit{FE}_j\})]\!]\iota &= ((\iota{\uparrow}1)[I_i \mapsto \mathcal{T}[\![T_i]\!]\iota], (\iota{\uparrow}2)[F_j \mapsto \mathcal{FE}[\![\mathit{FE}_j]\!]\iota])
\end{aligned}
$$

Figure 4.3 The factored semantics of L.

The factored semantics of L is presented in Fig. 4.3. Motivated by the needs of later sections, we retain the convention from §4.1 of working with the domain *Val* rather than *Int32*. Similarly, we also use *BVal* rather than *Bool*. The standard interpretations of binop_L, relop_L, and boolop_L are as one would expect, e.g., $v_1\ \mathit{binop}_L(\boxed{\oplus})\ v_2 = v_1\ \mathit{xor}\ v_2$, etc. The standard interpretations for $\mathit{lookupId}_{\mathit{std}}$ and $\mathit{lookupFuncId}_{\mathit{std}}$ select from the first and second components, respectively, of

a *LogicalStruct*: $\mathit{lookupId}_{\mathit{std}}\,\iota\,I = (\iota{\uparrow}1)(I)$ and $\mathit{lookupFuncId}_{\mathit{std}}\,\iota\,F = (\iota{\uparrow}2)(F)$. The standard interpretations for *access* and *update* select from, and store to, a map, respectively.

Let $U = (\{I_i \hookleftarrow T_i\}, \{F_j \hookleftarrow \mathit{FE}_j\})$. Because $\mathcal{U}[\![U]\!]\iota$ retains from ι the value of each constant I and function F for which an update is not defined explicitly in U (i.e., $I \in (\mathit{Id} - \{I_i\})$ and $F \in (\mathit{FuncId} - \{F_j\})$), as a notational convenience we sometimes treat U as if it contains an identity update for each such symbol; that is, we say that $(U{\uparrow}1)I = I$ for $I \in (\mathit{Id} - \{I_i\})$, and $(U{\uparrow}2)F = F$ for $F \in (\mathit{FuncId} - \{F_j\})$.

4.2.2 PL : A Simple Source-Level Language

PL is the language from §4.1, extended with some additional kinds of `int`-valued expressions, an address-generation expression, a dereferencing expression, and an indirect-assignment statement. Note that arithmetic operations can also occur inside a dereference expression; i.e., PL allows arithmetic to be performed on addresses (including bitwise operations on addresses: see Ex. 4.2).

$$c \in C_{\mathit{Int32}}, I \in \mathit{Id}, E \in \mathit{Expr}, \mathit{BE} \in \mathit{BoolExpr}, S \in \mathit{Stmt}$$

$$\begin{aligned} c &::= 0 \mid 1 \mid \ldots \\ E &::= c \mid I \mid \&I \mid *E \mid E_1\ \mathit{op2}\ E_2 \mid \mathit{BE}\ ?\ E_1 : E_2 \\ \mathit{BE} &::= \mathbb{T} \mid \mathbb{F} \mid E_1\ \mathit{rop}\ E_2 \mid \neg \mathit{BE}_1 \mid \mathit{BE}_1\ \mathit{bop}\ \mathit{BE}_2 \\ S &::= I = E; \mid *I = E; \mid S_1\ S_2 \end{aligned}$$

Semantics of PL. The factored semantics of PL is presented in Fig. 4.4. The semantic domain *Loc* stands for *locations* (or memory addresses). We identify *Loc* with the set *Val* of values. A state $\sigma \in \mathit{State}$ is a pair (η, ρ), where, in the standard semantics, *environment* $\eta \in \mathit{Env} = \mathit{Id} \to \mathit{Loc}$ maps identifiers to their associated locations and *store* $\rho \in \mathit{Store} = \mathit{Loc} \to \mathit{Val}$ maps each location to the value that it holds.

$$
\begin{aligned}
v &\in \mathit{Val} \\
l &\in \mathit{Loc} = \mathit{Val} \\
\eta &\in \mathit{Env} = \mathit{Id} \to \mathit{Loc} \\
\rho &\in \mathit{Store} = \mathit{Loc} \to \mathit{Val} \\
\sigma &\in \mathit{State} = \mathit{Store} \times \mathit{Env}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{E} &: \mathit{Expr} \to \mathit{State} \to \mathit{Val} \\
\mathcal{E}[\![c]\!]\sigma &= \mathit{const}(c) \\
\mathcal{E}[\![I]\!]\sigma &= \mathit{lookupState}\ \sigma\ I \\
\mathcal{E}[\![\&I]\!]\sigma &= \mathit{lookupEnv}\ \sigma\ I \\
\mathcal{E}[\![*E]\!]\sigma &= \mathit{lookupStore}\ \sigma\ (\mathcal{E}[\![E]\!]\sigma) \\
\mathcal{E}[\![E_1\ \mathit{op2}\ E_2]\!]\sigma &= \mathcal{E}[\![E_1]\!]\sigma\ \mathit{binop}(\mathit{op2})\ \mathcal{E}[\![E_2]\!]\sigma \\
\mathcal{E}[\![\mathit{BE}\ ?\ E_1 : E_2]\!]\sigma &= \mathit{cond}(\mathcal{B}[\![\mathit{BE}]\!]\sigma, \mathcal{E}[\![E_1]\!]\sigma, \mathcal{E}[\![E_2]\!]\sigma)
\end{aligned}
$$

$$
\begin{aligned}
\mathit{const} &: C_{\mathit{Int32}} \to \mathit{Val} \\
\mathit{cond} &: \mathit{BVal} \to \mathit{Val} \to \mathit{Val} \to \mathit{Val} \\
\mathit{lookupState} &: \mathit{State} \to \mathit{Id} \to \mathit{Val} \\
\mathit{lookupEnv} &: \mathit{State} \to \mathit{Id} \to \mathit{Loc} \\
\mathit{lookupStore} &: \mathit{State} \to \mathit{Loc} \to \mathit{Val} \\
\mathit{updateStore} &: \mathit{State} \to \mathit{Loc} \to \mathit{Val} \to \mathit{State}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{B} &: \mathit{BoolExpr} \to \mathit{State} \to \mathit{BVal} \\
\mathcal{B}[\![\mathbb{T}]\!]\sigma &= \mathbb{T} \\
\mathcal{B}[\![\mathbb{F}]\!]\sigma &= \mathbb{F} \\
\mathcal{B}[\![E_1\ \mathit{rop}\ E_2]\!]\sigma &= \mathcal{E}[\![E_1]\!]\sigma\ \mathit{relop}(\mathit{rop})\ \mathcal{E}[\![E_2]\!]\sigma \\
\mathcal{B}[\![\neg \mathit{BE}_1]\!]\sigma &= \neg\mathcal{B}[\![\mathit{BE}_1]\!]\sigma \\
\mathcal{B}[\![\mathit{BE}_1\ \mathit{bop}\ \mathit{BE}_2]\!]\sigma &= \mathcal{B}[\![\mathit{BE}_1]\!]\sigma\ \mathit{boolop}(\mathit{bop})\ \mathcal{B}[\![\mathit{BE}_2]\!]\sigma
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{I} &: \mathit{Stmt} \to \mathit{State} \to \mathit{State} \\
\mathcal{I}[\![I = E;]\!]\sigma &= \mathit{updateStore}\ \sigma\ (\mathit{lookupEnv}\ \sigma\ I)\ (\mathcal{E}[\![E]\!]\sigma) \\
\mathcal{I}[\![*I = E;]\!]\sigma &= \mathit{updateStore}\ \sigma\ (\mathcal{E}[\![I]\!]\sigma)\ (\mathcal{E}[\![E]\!]\sigma) \\
\mathcal{I}[\![S_1\ S_2]\!]\sigma &= \mathcal{I}[\![S_2]\!](\mathcal{I}[\![S_1]\!]\sigma)
\end{aligned}
$$

Figure 4.4 The factored semantics of PL.

The standard interpretations of the operators used in the PL semantics are

$$
\begin{aligned}
\mathit{BVal}_{\mathit{std}} &= \mathit{BVal} \\
\mathit{Val}_{\mathit{std}} &= \mathit{Int32} \\
\mathit{Loc}_{\mathit{std}} &= \mathit{Int32} \\
\eta \in \mathit{Env}_{\mathit{std}} &= \mathit{Id} \to \mathit{Loc}_{\mathit{std}} \\
\rho \in \mathit{Store}_{\mathit{std}} &= \mathit{Loc}_{\mathit{std}} \to \mathit{Val}_{\mathit{std}}
\end{aligned}
$$

$$\begin{aligned}
\mathit{cond}_{\mathit{std}} &= \lambda b.\lambda v_1.\lambda v_2.\,(b\,?\,v_1 : v_2)\\
\mathit{lookupState}_{\mathit{std}} &= \lambda(\eta,\rho).\lambda I.\rho(\eta(I))\\
\mathit{lookupEnv}_{\mathit{std}} &= \lambda(\eta,\rho).\lambda I.\eta(I)\\
\mathit{lookupStore}_{\mathit{std}} &= \lambda(\eta,\rho).\lambda l.\rho(l)\\
\mathit{updateStore}_{\mathit{std}} &= \lambda(\eta,\rho).\lambda l.\lambda v.(\eta,\rho[l \mapsto v])
\end{aligned}$$

Handling Computations that "Go Wrong". In accounts of axiomatic semantics [146] and relational semantics [171], one generally considers four outcomes of an execution: an execution *terminates* (in some final state), *goes wrong*, *blocks*, or *diverges*. Because we are only providing the semantics of individual statements/instructions, to simplify matters, we consider only semantic specifications that are terminating. This eliminates outcomes that block or diverge.

We sidestep the need for an explicit outcome for "goes wrong" by introducing an additional *BVal* variable in the state, `isRunning`, which is set to false to model computations that "go wrong". In the extended semantics, a state $\sigma \in \mathit{State}$ is a triple $(\eta, \rho, \texttt{isRunning})$. Fig. 4.5 shows a sketch of how to add the semantics of the outcome for "divide-by-zero". For the moment, we consider only deterministic specifications. §4.6 discusses how we handle non-determinism.

4.2.3 MC: A Simple Machine-Code Language

MC is based on the x86 instruction set, but greatly simplified to have just four registers, one flag, and four instructions.

$$\begin{aligned}
r \in \mathit{register}, do \in \mathit{dst_operand},&\\
so \in \mathit{src_operand}, i \in \mathit{instruction}&\\
r &::= \texttt{eax} \mid \texttt{ebx} \mid \texttt{ebp} \mid \texttt{eip}\\
\mathit{flagName} &::= \texttt{zf}\\
do &::= \mathit{Indirect}(r, \mathit{Val}) \mid \mathit{DirectReg}(r)\\
so &::= do \cup \mathit{Immediate}(\mathit{Val})\\
\mathit{instruction} &::= \texttt{mov}(do, so) \mid \texttt{cmp}(do, so)\\
&\quad \mid \mathit{XOR}(do, so) \mid \texttt{jz}(do)
\end{aligned}$$

$$
\begin{array}{rcl}
\mathit{Loc} & = & \mathit{Val} \\
\mathit{Env} & = & \mathit{Id} \to \mathit{Loc} \\
\mathit{Store} & = & \mathit{Loc} \to \mathit{Val} \\
\mathit{State} & = & \mathit{Store} \times \mathit{Env} \times \mathit{BVal} \\
 & & \\
\mathit{const} & : & C_{\mathit{Int32}} \to \mathit{Val} \\
\mathit{lookupState} & : & \mathit{State} \to \mathit{Id} \to \mathit{Val} \\
\mathit{getIsRunning} & : & (\mathit{Val}, \mathit{BVal}) \to \mathit{BVal} \\
\mathit{lookupIsRunning} & : & \mathit{State} \to \mathit{BVal} \\
\mathit{updateIsRunning} & : & \mathit{State} \to \mathit{BVal} \to \mathit{State} \\
\mathit{getIsRunning} & = & \lambda(v, b).b \\
\mathit{lookupIsRunning} & = & \lambda(\eta, \rho, b).b \\
\mathit{updateIsRunning} & = & \lambda(\eta, \rho, b).\lambda b'.(\eta, \rho, b')
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{E} : \mathit{Expr} \to \mathit{State} \to (\mathit{Val}, \mathit{BVal}) & & \\
\mathcal{E}[\![c]\!]\sigma & = & (\mathit{const}(c),\ \mathbb{T}) \\
\mathcal{E}[\![I]\!]\sigma & = & (\mathit{lookupState}\ \sigma\ I,\ \mathbb{T}) \\
\mathcal{E}[\![E_1/E_2]\!]\sigma & = & (\mathcal{E}[\![E_2]\!]\sigma = 0) \\
 & & \quad ?\ (1,\ \mathbb{F}) \\
 & & \quad :\ (\mathcal{E}[\![E_1]\!]\sigma / \mathcal{E}[\![E_2]\!]\sigma,\ \mathbb{T}) \\
 & & \\
\mathcal{I} : \mathit{Stmt} \to \mathit{State} \to \mathit{State} & & \\
\mathcal{I}[\![I = E;]\!]\sigma & = & (\mathit{lookupIsRunning}\ \sigma) = \mathbb{T} \\
 & & \quad ?\ (\mathit{getIsRunning}\ \mathcal{E}[\![E]\!]\sigma) = \mathbb{T} \\
 & & \quad\quad ?\ \mathit{updateStore}\ \sigma\ (\mathit{lookupEnv}\ \sigma\ I)\ (\mathcal{E}[\![E]\!]\sigma) \\
 & & \quad\quad :\ \mathit{updateIsRunning}\ \sigma\ \mathbb{F} \\
 & & \quad :\ \sigma
\end{array}
$$

Figure 4.5 An extended semantics of PL to accommodate the outcome of "divide-by-zero" execution.

Semantics of MC. The factored semantics of MC is presented in Fig. 4.6. It is similar to the semantics of PL, although MC exhibits two features not part of PL: there is an explicit program counter (`eip`), and MC includes the typical feature of machine-code languages that a branch is split across two instructions (`cmp ... jz`). An MC state $\sigma \in \mathit{State}$ is a triple $(\mathit{mem}, \mathit{reg}, \mathit{flag})$, where *mem* is a map *Val* $\to$ *Val*, *reg* is a map *register* $\to$ *Val*, and *flag* is a map *flagName* $\to$ *BVal*. We assume that each instruction is 4 bytes long; hence, the execution of a `mov`, `cmp`, or *XOR* increments the program-counter register `eip` by 4. `cmp` sets the value of `zf` according to the difference of the values of the two operands; `jz` updates `eip` depending on the value of flag `zf`.

4.3 Symbolic Analysis for PL via Reinterpretation

A PL state (η, ρ) can be modeled in $L[\mathrm{PL}]$ by using a function symbol F_ρ for store ρ, and a constant symbol $c_x \in \mathit{Id}$ for each PL identifier x. (To reduce clutter, we will use `x` for such constants instead of c_x.) Given $\iota \in \mathit{LogicalStruct}$, the constant symbols and their interpretations in ι correspond to environment η, and the interpretation of F_ρ in ι corresponds to store ρ.

$$
\begin{array}{rl}
\mathit{const} : C_{\mathit{Int32}} \to \mathit{Val} & \mathit{cond} \ : \ \mathit{BVal} \to \mathit{Val} \to \mathit{Val} \to \mathit{Val} \\
\mathit{lookup}_{\mathit{reg}} : \mathit{State} \to \mathit{register} \to \mathit{Val} & \mathit{lookup}_{\mathit{mem}} \ : \ \mathit{State} \to \mathit{Val} \to \mathit{Val} \\
\mathit{store}_{\mathit{reg}} : \mathit{State} \to \mathit{register} \to \mathit{Val} \to \mathit{State} & \mathit{store}_{\mathit{mem}} \ : \ \mathit{State} \to \mathit{Val} \to \mathit{Val} \to \mathit{State} \\
\mathit{lookup}_{\mathit{flag}} : \mathit{State} \to \mathit{flagName} \to \mathit{BVal} & \mathit{incr}_{\mathtt{eip}} \ : \ \mathit{State} \to \mathit{State} \\
\mathit{store}_{\mathit{flag}} : \mathit{State} \to \mathit{flagName} \to \mathit{BVal} \to \mathit{State} & \mathit{incr}_{\mathtt{eip}} = \lambda\sigma.\mathit{store}_{\mathit{reg}}(\sigma, \mathtt{eip}, \mathcal{R}[\![\mathtt{eip}]\!]\sigma \ \mathit{binop}(+)\ 4)
\end{array}
$$

$$
\begin{array}{rl}
\mathcal{R} : \mathit{reg} \to \mathit{State} \to \ \mathit{Val} & \mathcal{O} : \mathit{src\_operand} \to \mathit{State} \to \mathit{Val} \\
\mathcal{R}[\![r]\!]\sigma \ = \ \mathit{lookup}_{\mathit{reg}}(\sigma, r) & \mathcal{O}[\![\mathit{Indirect}(r, c)]\!]\sigma \ = \ \mathit{lookup}_{\mathit{mem}}(\sigma, \mathcal{R}[\![r]\!]\sigma \ \mathit{binop}(+)\ \mathit{const}(c)) \\
\mathcal{K} : \mathit{flagName} \to \mathit{State} \to \mathit{BVal} & \mathcal{O}[\![\mathit{DirectReg}(r)]\!]\sigma \ = \ \mathcal{R}[\![r]\!]\sigma \\
\mathcal{K}[\![\mathtt{zf}]\!]\sigma \ = \ \mathit{lookup}_{\mathit{flag}}(\sigma, \mathtt{zf}) & \mathcal{O}[\![\mathit{Immediate}(c)]\!]\sigma \ = \ \mathit{const}(c)
\end{array}
$$

$$
\begin{array}{rl}
\multicolumn{2}{c}{\mathcal{I} : \mathit{instruction} \to \mathit{State} \to \mathit{State}} \\
\mathcal{I}[\![\mathtt{mov}(\mathit{Indirect}(r, c), so)]\!]\sigma & = \ \mathit{incr}_{\mathtt{eip}}(\mathit{store}_{\mathit{mem}}(\sigma, \mathcal{R}[\![r]\!]\sigma \ \mathit{binop}(+)\ \mathit{const}(c), \mathcal{O}[\![so]\!]\sigma)) \\
\mathcal{I}[\![\mathtt{mov}(\mathit{DirectReg}(r), so)]\!]\sigma & = \ \mathit{incr}_{\mathtt{eip}}(\mathit{store}_{\mathit{reg}}(\sigma, r, \mathcal{O}[\![so]\!]\sigma)) \\
\mathcal{I}[\![\mathtt{cmp}(do, so)]\!]\sigma & = \ \mathit{incr}_{\mathtt{eip}}(\mathit{store}_{\mathit{flag}}(\sigma, \mathtt{zf}, \mathcal{O}[\![do]\!]\sigma \ \mathit{binop}(-)\ \mathcal{O}[\![so]\!]\sigma \ \mathit{relop}(=)\ 0)) \\
\mathcal{I}[\![\mathit{XOR}(do{:}\mathit{Indirect}(r, c), so)]\!]\sigma & = \ \mathit{incr}_{\mathtt{eip}}(\mathit{store}_{\mathit{mem}}(\sigma, \mathcal{R}[\![r]\!]\sigma \ \mathit{binop}(+)\ \mathit{const}(c), \mathcal{O}[\![do]\!]\sigma \ \mathit{binop}(\oplus)\ \mathcal{O}[\![so]\!]\sigma)) \\
\mathcal{I}[\![\mathit{XOR}(do{:}\mathit{DirectReg}(r), so)]\!]\sigma & = \ \mathit{incr}_{\mathtt{eip}}(\mathit{store}_{\mathit{reg}}(\sigma, r, \mathcal{O}[\![do]\!]\sigma \ \mathit{binop}(\oplus)\ \mathcal{O}[\![so]\!]\sigma)) \\
\mathcal{I}[\![\mathtt{jz}(do)]\!]\sigma & = \ \mathit{store}_{\mathit{reg}}(\sigma, \mathtt{eip}, \mathit{cond}(\mathcal{K}[\![\mathtt{zf}]\!]\sigma, \mathcal{O}[\![do]\!]\sigma, \mathcal{R}[\![\mathtt{eip}]\!]\sigma \ \mathit{binop}(+)\ 4))
\end{array}
$$

Figure 4.6 The factored semantics of MC.

Symbolic Evaluation. A primitive for forward symbolic-evaluation must solve the following problem: *Given the semantic definition of a programming language, together with a specific statement s, create a logical formula that captures the semantics of s.* The following table illustrates how the semantics of PL statements can be expressed as $L[\mathrm{PL}]$ structure-update expressions:

PL	$L[\mathrm{PL}]$
$x = 17;$	$(\emptyset, \{F'_\rho \hookleftarrow F_\rho[\mathtt{x} \mapsto 17]\})$
$x = y;$	$(\emptyset, \{F'_\rho \hookleftarrow F_\rho[\mathtt{x} \mapsto F_\rho(\mathtt{y})]\})$
$x = *q;$	$(\emptyset, \{F'_\rho \hookleftarrow F_\rho[\mathtt{x} \mapsto F_\rho(F_\rho(\mathtt{q}))]\})$

To create such expressions automatically using semantic reinterpretation, we use formulas of the logic $L[\mathrm{PL}]$ as a reinterpretation domain for the meta-language primitives used to define PL. The

base types and the state type of the meta-language are reinterpreted as follows (our convention is to mark each reinterpreted base type, function type, and operator with an overbar): $\overline{\textit{Val}} = \textit{Term}$, $\overline{\textit{BVal}} = \textit{Formula}$, and $\overline{\textit{State}} = \textit{StructUpdate}$. The operators used in PL's meaning functions $\mathcal{E}$, $\mathcal{B}$, and $\mathcal{I}$ are reinterpreted over these domains as follows:

- The arithmetic, bitwise, relational, and logical operators are interpreted as syntactic constructors of $L[\text{PL}]$ *Term*s and *Formula*s, e.g.,

  $$\overline{\textit{binop}}(\oplus) = \lambda T_1.\lambda T_2.T_1 \boxed{\oplus} T_2.$$

  Straightforward simplifications are also performed; e.g., $0 \boxed{\oplus} a$ simplifies to a, etc. Other simplifications that we perform are similar to ones used by others, such as the preprocessing steps used in decision procedures (e.g., the ite-lifting and read-over-write transformations for operations on functions [89]).
- $\overline{\textit{cond}}$ residuates an $\textit{ite}(\cdot,\cdot,\cdot)$ *Term* when the result cannot be simplified to a single branch.

The other operations used in the PL semantics are reinterpreted as follows:

$$\begin{aligned}
\overline{\textit{lookupState}} &: \textit{StructUpdate} \rightarrow \textit{Id} \rightarrow \textit{Term} \\
\overline{\textit{lookupState}} &= \lambda U.\lambda I.((U{\uparrow}2)F_\rho)((U{\uparrow}1)I) \\
\overline{\textit{lookupEnv}} &: \textit{StructUpdate} \rightarrow \textit{Id} \rightarrow \textit{Term} \\
\overline{\textit{lookupEnv}} &= \lambda U.\lambda I.(U{\uparrow}1)I \\
\overline{\textit{lookupStore}} &: \textit{StructUpdate} \rightarrow \textit{Term} \rightarrow \textit{Term} \\
\overline{\textit{lookupStore}} &= \lambda U.\lambda T.((U{\uparrow}2)F_\rho)(T) \\
\overline{\textit{updateStore}} &: \textit{StructUpdate} \rightarrow \textit{Term} \rightarrow \textit{Term} \\
&\qquad \rightarrow \textit{StructUpdate} \\
\overline{\textit{updateStore}} &= \lambda U.\lambda T_1.\lambda T_2.((U{\uparrow}1),(U{\uparrow}2)[F_\rho \mapsto ((U{\uparrow}2)F_\rho)[T_1 \mapsto T_2]])
\end{aligned}$$

By extension, this produces functions $\overline{\mathcal{E}}$, $\overline{\mathcal{B}}$, and $\overline{\mathcal{I}}$ with the types shown in Fig. 4.7.

In particular, given a *StructUpdate* U, function $\overline{\mathcal{I}}$ translates a statement s of PL to the *StructUpdate* $\overline{\mathcal{I}}[\![s]\!]U$ in logic $L[\text{PL}]$. To perform symbolic evaluation along a path π, one starts with the *StructUpdate* $U_{id} = (\emptyset, \{F'_\rho \hookleftarrow F_\rho\})$ and repeatedly calls function $\overline{\mathcal{I}}$ with the next statement in π and the current *StructUpdate*.

Standard	Reinterpreted
$\mathcal{E}$: *Expr* $\rightarrow$ *State* $\rightarrow$ *Val*	$\overline{\mathcal{E}}$: *Expr* $\rightarrow$ *StructUpdate* $\rightarrow$ *Term*
$\mathcal{B}$: *BoolExpr* $\rightarrow$ *State* $\rightarrow$ *BVal*	$\overline{\mathcal{B}}$: *BoolExpr* $\rightarrow$ *StructUpdate* $\rightarrow$ *Formula*
$\mathcal{I}$: *Stmt* $\rightarrow$ *State* $\rightarrow$ *State*	$\overline{\mathcal{I}}$: *Stmt* $\rightarrow$ *StructUpdate* $\rightarrow$ *StructUpdate*

Figure 4.7 Standard types of the PL meaning functions, and the reinterpreted types used to obtain an implementation of symbolic evaluation.

$$
\begin{aligned}
\overline{\mathcal{I}}[\![x = x \oplus y;]\!]U_{id} &= (\emptyset, \{F'_\rho \hookleftarrow F_\rho[\mathtt{x} \mapsto (\overline{\mathcal{E}}[\![x]\!]U_{id} \boxed{\oplus} \overline{\mathcal{E}}[\![y]\!]U_{id})]\}) \\
&= (\emptyset, \{F'_\rho \hookleftarrow F_\rho[\mathtt{x} \mapsto (F_\rho(\mathtt{x}) \boxed{\oplus} F_\rho(\mathtt{y}))]\}) = U_1 \\
\overline{\mathcal{I}}[\![y = x \oplus y;]\!]U_1 &= (\emptyset, \{F'_\rho \hookleftarrow F_\rho[\mathtt{x} \mapsto (F_\rho(\mathtt{x}) \boxed{\oplus} F_\rho(\mathtt{y}))][\mathtt{y} \mapsto (\overline{\mathcal{E}}[\![x]\!]U_1 \boxed{\oplus} \overline{\mathcal{E}}[\![y]\!]U_1)]\}) \\
&= (\emptyset, \{F'_\rho \hookleftarrow F_\rho[\mathtt{x} \mapsto (F_\rho(\mathtt{x}) \boxed{\oplus} F_\rho(\mathtt{y}))][\mathtt{y} \mapsto ((F_\rho(\mathtt{x}) \boxed{\oplus} F_\rho(\mathtt{y})) \boxed{\oplus} F_\rho(\mathtt{y}))]\}) \\
&= (\emptyset, \{F'_\rho \hookleftarrow F_\rho[\mathtt{x} \mapsto (F_\rho(\mathtt{x}) \boxed{\oplus} F_\rho(\mathtt{y}))][\mathtt{y} \mapsto F_\rho(\mathtt{x})]\}) = U_2 \\
\overline{\mathcal{I}}[\![x = x \oplus y;]\!]U_2 &= (\emptyset, \{F'_\rho \hookleftarrow F_\rho[\mathtt{x} \mapsto (\overline{\mathcal{E}}[\![x]\!]U_2 \boxed{\oplus} \overline{\mathcal{E}}[\![y]\!]U_2)][\mathtt{y} \mapsto F_\rho(\mathtt{x})]\}) \\
&= (\emptyset, \{F'_\rho \hookleftarrow F_\rho[\mathtt{x} \mapsto ((F_\rho(\mathtt{x}) \boxed{\oplus} F_\rho(\mathtt{y})) \boxed{\oplus} F_\rho(\mathtt{x}))][\mathtt{y} \mapsto F_\rho(\mathtt{x})]\}) \\
&= (\emptyset, \{F'_\rho \hookleftarrow F_\rho[\mathtt{x} \mapsto F_\rho(\mathtt{y})][\mathtt{y} \mapsto F_\rho(\mathtt{x})]\}) = U_{swap}
\end{aligned}
$$

Figure 4.8 Symbolic evaluation of Fig. 4.1(a) via semantic reinterpretation, starting with the *StructUpdate* $U_{id} = (\emptyset, \{F'_\rho \hookleftarrow F_\rho\})$.

Example 4.2 The steps of symbolic evaluation of Fig. 4.1(a) via semantic reinterpretation, starting with U_{id}, are shown in Fig. 4.8. The resulting *StructUpdate*, U_{swap}, can be considered to be the 2-vocabulary formula

$$F'_\rho = F_\rho[\mathtt{x} \mapsto F_\rho(\mathtt{y})][\mathtt{y} \mapsto F_\rho(\mathtt{x})],$$

which expresses a state change in which the values of program variables x and y are swapped.

Algebraic simplification plays an important role. For example, when y is updated in U_1 by

$$[\mathtt{y} \mapsto ((F_\rho(\mathtt{x}) \boxed{\oplus} F_\rho(\mathtt{y})) \boxed{\oplus} F_\rho(\mathtt{y}))]$$

(see Fig. 4.8), the update is simplified to $[\mathtt{y} \mapsto F_\rho(\mathtt{x})]$. □

$$
\begin{aligned}
U_1 &= (\emptyset, \{F'_\rho \hookleftarrow F_\rho[0 \mapsto v][\mathit{px} \mapsto \mathit{py}][\mathit{py} \mapsto \mathit{py}]\}) \\
\overline{\mathcal{I}}[\![*px = *px \oplus *py;]\!]U_1 &= (\emptyset, \{F'_\rho \hookleftarrow F_\rho[0 \mapsto v][\mathit{px} \mapsto \mathit{py}][\mathit{py} \mapsto (\overline{\mathcal{E}}[\![*px]\!]U_1 \boxed{\oplus} \overline{\mathcal{E}}[\![*py]\!]U_1)]\}) \\
&= (\emptyset, \{F'_\rho \hookleftarrow F_\rho[0 \mapsto v][\mathit{px} \mapsto \mathit{py}][\mathit{py} \mapsto (\mathit{py} \boxed{\oplus} \mathit{py})]\}) \\
&= (\emptyset, \{F'_\rho \hookleftarrow F_\rho[0 \mapsto v][\mathit{px} \mapsto \mathit{py}][\mathit{py} \mapsto 0]\}) = U_2 \\
\overline{\mathcal{I}}[\![*py = *px \oplus *py;]\!]U_2 &= (\emptyset, \{F'_\rho \hookleftarrow F_\rho[0 \mapsto (\overline{\mathcal{E}}[\![*px]\!]U_2 \boxed{\oplus} \overline{\mathcal{E}}[\![*py]\!]U_2)][\mathit{px} \mapsto \mathit{py}][\mathit{py} \mapsto 0]\}) \\
&= (\emptyset, \{F'_\rho \hookleftarrow F_\rho[0 \mapsto (0 \boxed{\oplus} v)][\mathit{px} \mapsto \mathit{py}][\mathit{py} \mapsto 0]\}) \\
&= (\emptyset, \{F'_\rho \hookleftarrow F_\rho[0 \mapsto v][\mathit{px} \mapsto \mathit{py}][\mathit{py} \mapsto 0]\}) = U_3 \\
\overline{\mathcal{I}}[\![*px = *px \oplus *py;]\!]U_3 &= (\emptyset, \{F'_\rho \hookleftarrow F_\rho[0 \mapsto v][\mathit{px} \mapsto \mathit{py}][\mathit{py} \mapsto (\overline{\mathcal{E}}[\![*px]\!]U_3 \boxed{\oplus} \overline{\mathcal{E}}[\![*py]\!]U_3)]\}) \\
&= (\emptyset, \{F'_\rho \hookleftarrow F_\rho[0 \mapsto v][\mathit{px} \mapsto \mathit{py}][\mathit{py} \mapsto (0 \boxed{\oplus} v)]\}) \\
&= (\emptyset, \{F'_\rho \hookleftarrow F_\rho[0 \mapsto v][\mathit{px} \mapsto \mathit{py}][\mathit{py} \mapsto v]\}) = U_4
\end{aligned}
$$

Figure 4.9 Symbolic evaluation of Fig. 4.1(b) via semantic reinterpretation, starting with a *StructUpdate* that corresponds to the "Before" column of Fig. 4.1(c).

Example 4.3 To illustrate symbolic evaluation for an example that involves pointers and pointer-dereferencing operations, Fig. 4.9 shows the steps of symbolic evaluation of Fig. 4.1(b) via semantic reinterpretation, starting with a *StructUpdate* that corresponds to the "Before" column of Fig. 4.1(c). The program from Fig. 4.1(b) works correctly when there is no aliasing; however, it does not always work correctly when started from the kind of state shown in the "Before" column of Fig. 4.1(c). The *StructUpdate* U_4 obtained via our symbolic-evaluation primitive can be considered to be the 2-vocabulary formula

$$F'_\rho = F_\rho[0 \mapsto v][\mathit{px} \mapsto \mathit{py}][\mathit{py} \mapsto v],$$

which expresses a state change that does not usually perform a successful swap. The example shows that the symbolic-evaluation method can faithfully track non-trivial situations that involve pointer aliasing. □

The correctness of our method for performing symbolic evaluation is captured by the following theorem:

Theorem 4.4 For all $s \in$ *Stmt*, $U \in$ *StructUpdate*, and $\iota \in$ *LogicalStruct*, the meaning of $\overline{\mathcal{I}}[\![s]\!]U$ in ι (i.e., $\mathcal{U}[\![\overline{\mathcal{I}}[\![s]\!]U]\!]\iota$) is equivalent to running $\mathcal{I}$ on s with an input state obtained from $\mathcal{U}[\![U]\!]\iota$. That is,

$$\mathcal{U}[\![\overline{\mathcal{I}}[\![s]\!]U]\!]\iota = \mathcal{I}[\![s]\!](\mathcal{U}[\![U]\!]\iota).$$

Proof: See App. B.1. □

WLP. $\mathcal{WLP}(s, \varphi)$ characterizes the set of states σ such that the execution of s starting in σ either fails to terminate or results in a state σ' such that $\varphi(\sigma')$ holds. For a language that only has `int`-valued variables, the $\mathcal{WLP}$ of a postcondition (specified by formula φ) with respect to an assignment statement *var* $=$ *rhs*; can be expressed as the formula obtained by substituting *rhs* for all (free) occurrences of *var* in φ: φ[*var* $\leftarrow$ *rhs*].

For a language with pointer variables, such as PL, syntactic substitution is not adequate for finding $\mathcal{WLP}$ formulas. For instance, suppose that we are interested in finding a formula for the $\mathcal{WLP}$ of postcondition $x = 5$ with respect to $*p = e;$. It is not correct merely to perform the substitution $(x = 5)[*p \leftarrow e]$. That substitution yields $x = 5$, whereas the $\mathcal{WLP}$ depends on the execution context in which $*p = e;$ is evaluated:

- If p points to x, then the $\mathcal{WLP}$ formula should be $e = 5$.
- If p does not point to x, then the $\mathcal{WLP}$ formula should be $x = 5$.

The desired formula can be expressed informally as

$$((p = \&x)\ ?\ e : x) = 5.$$

For a program fragment that involves multiple pointer variables, the $\mathcal{WLP}$ formula may have to take into account all possible aliasing combinations. This is the essence of Morris's rule of substitution [138]. One of the most important features of our approach is its ability to create correct implementations of Morris's rule of substitution automatically—and basically for free.

Example 4.5 In $L[\text{PL}]$, such a formula would be expressed as shown in the lower row below.

Informal	$\mathcal{WLP}(*p = e, x = 5) = ((p = \&x)\ ?\ e : x) = 5$
$L[\text{PL}]$	$\mathcal{WLP}(*p = e, F_\rho(\texttt{x}) \boxed{=} 5) = \mathit{ite}(F_\rho(\texttt{p}) \boxed{=} \texttt{x}, F_\rho(\texttt{e}), F_\rho(\texttt{x})) \boxed{=} 5$

In Ex. 4.7, we will show how the latter formula is created via semantic reinterpretation. □

To create primitives for $\mathcal{WLP}$ and symbolic composition via semantic reinterpretation, we again use $L[\text{PL}]$ as a reinterpretation domain; however, there is a trick: in contrast with what is done to generate symbolic-evaluation primitives, we use the *StructUpdate* type of $L[\text{PL}]$ to reinterpret the meaning functions $\mathcal{U}$, $\mathcal{FE}$, $\mathcal{F}$, and $\mathcal{T}$ of $L[\text{PL}]$ itself! By this means, the "alternative meaning" of a *Term*/*Formula*/*FuncExpr*/*StructUpdate* is a (usually different) *Term*/*Formula*/*FuncExpr*/*StructUpdate* in which some substitution and/or simplification has taken place. The general scheme is outlined in the following table:

Meaning Functions	Type Reinterpreted	Replacement Type	Function Created
$\mathcal{I}, \mathcal{E}, \mathcal{B}$	*State*	*StructUpdate*	Symbolic evaluation
$\mathcal{F}, \mathcal{T}$	*LogicalStruct*	*StructUpdate*	$\mathcal{WLP}$
$\mathcal{U}, \mathcal{FE}, \mathcal{F}, \mathcal{T}$	*LogicalStruct*	*StructUpdate*	Symbolic composition

In §4.2.1, we defined the semantics of $L[\cdot]$ in a form that would make it amenable to semantic reinterpretation. However, one small point needs adjustment: in §4.2.1, the type signatures of *LogicalStruct*, *lookupFuncId*, *access*, *update*, and $\mathcal{FE}$ include occurrences of *Val* $\rightarrow$ *Val*. This was done to make the types more intuitive; however, for reinterpretation to work, an additional level of factoring is necessary. In particular, the occurrences of *Val* $\rightarrow$ *Val* need to be replaced by *FVal*. The standard semantics of *FVal* is *Val* $\rightarrow$ *Val*; however, for creating symbolic-analysis primitives, *FVal* is reinterpreted as *FuncExpr*.

The reinterpretation used for $\mathcal{U}$, $\mathcal{FE}$, $\mathcal{F}$, and $\mathcal{T}$ is similar to what was used for symbolic evaluation of PL programs:

- $\overline{\textit{Val}} = \textit{Term}$, $\overline{\textit{BVal}} = \textit{Formula}$, $\overline{\textit{FVal}} = \textit{FuncExpr}$, and $\overline{\textit{LogicalStruct}} = \textit{StructUpdate}$.
- The arithmetic, bitwise, relational, and logical operators are interpreted as syntactic *Term* and *Formula* constructors of L, e.g.,

$$\overline{\textit{binop}_L}(\boxed{\oplus}) = \lambda T_1.\lambda T_2.T_1 \boxed{\oplus} T_2,$$

  although straightforward simplifications are also performed.
- $\overline{\textit{cond}_L}$ residuates an *ite*$(\cdot,\cdot,\cdot)$ *Term* when the result cannot be simplified to a single branch.
- $\overline{\textit{lookupId}}$ and $\overline{\textit{lookupFuncId}}$ are resolved immediately, rather than residuated:
  - $\overline{\textit{lookupId}}\ (\{I_i \hookleftarrow T_i\}, \{F_j \hookleftarrow \textit{FE}_j\})\ I_k = T_k$
  - $\overline{\textit{lookupFuncId}}\ (\{I_i \hookleftarrow T_i\}, \{F_j \hookleftarrow \textit{FE}_j\})\ F_k = \textit{FE}_k$.
- $\overline{\textit{access}}$ and $\overline{\textit{update}}$ are discussed below.

By extension, this produces reinterpreted meaning functions $\overline{\mathcal{U}}$, $\overline{\mathcal{FE}}$, $\overline{\mathcal{F}}$, and $\overline{\mathcal{T}}$.

Somewhat surprisingly, we do not need to introduce an explicit operation of substitution for our logic because *a substitution operation is produced as a by-product of reinterpretation*. In particular, in the standard semantics for L, the return types of meaning function $\mathcal{T}$ and helper function *lookupId* are both *Val*. However, in the reinterpreted semantics, a $\overline{\textit{Val}}$ is a *Term*—i.e., something *symbolic*—which is used in subsequent computations. Thus, when $\iota \in \textit{LogicalStruct}$ is reinterpreted as $U \in \textit{StructUpdate}$, the reinterpretation of formula φ via $\overline{\mathcal{F}}[\![\varphi]\!]U$ substitutes *Term*s found in U into φ: $\overline{\mathcal{F}}[\![\varphi]\!]U$ calls $\overline{\mathcal{T}}[\![T]\!]U$, which may call $\overline{\textit{lookupId}}\ U\ I$; the latter would return a *Term* fetched from U, which would be a subterm of the answer returned by $\overline{\mathcal{T}}[\![T]\!]U$, which in turn would be a subterm of the answer returned by $\overline{\mathcal{F}}[\![\varphi]\!]U$.

To create a formula for $\mathcal{WLP}$ via semantic reinterpretation, we make use of both $\overline{\mathcal{F}}$, the reinterpreted logic semantics, and $\overline{\mathcal{I}}$, the reinterpreted programming-language semantics. The $\mathcal{WLP}$ formula for φ with respect to statement s is obtained by performing the following computation:

$$\mathcal{WLP}(s,\varphi) = \overline{\mathcal{F}}[\![\varphi]\!](\overline{\mathcal{I}}[\![s]\!]U_{id}). \tag{4.1}$$

Example 4.6 In Ex. 4.2 and Fig. 4.8, we derived the following *StructUpdate*, which expresses in $L[\mathrm{PL}]$ the semantics of the swap-code fragment *swap* from Fig. 4.1(a):

$$\begin{aligned} U_{swap} &= \overline{\mathcal{I}}[\![swap]\!]U_{id} \\ &= (\emptyset, \{F'_\rho \hookleftarrow F_\rho[\mathtt{x} \mapsto F_\rho(\mathtt{y})][\mathtt{y} \mapsto F_\rho(\mathtt{x})]\}). \end{aligned}$$

Using the method given in Eqn. (4.1), we obtain the following *Formula* of $L[\mathrm{PL}]$ for $\mathcal{WLP}(swap, F_\rho(\mathtt{x}) \boxed{=} 2)$:

$$\begin{aligned} &\mathcal{WLP}(swap, F_\rho(\mathtt{x}) \boxed{=} 2) \\ &\quad= \overline{\mathcal{F}}[\![F_\rho(\mathtt{x}) \boxed{=} 2]\!]U_{swap} \\ &\quad= (\overline{\mathcal{T}}[\![F_\rho(\mathtt{x})]\!]U_{swap}) \boxed{=} (\overline{\mathcal{T}}[\![2]\!]U_{swap}) \\ &\quad= (\overline{access}(\overline{\mathcal{FE}}[\![F_\rho]\!]U_{swap}, \overline{\mathcal{T}}[\![\mathtt{x}]\!]U_{swap})) \boxed{=} (\overline{const}(2)) \\ &\quad= \left(\overline{access}\left(\begin{array}{l}\overline{lookupFuncId}\ U_{swap}\ F_\rho, \\ \overline{lookupId}\ U_{swap}\ \mathtt{x}\end{array}\right)\right) \boxed{=} 2 \\ &\quad= (\overline{access}(F_\rho[\mathtt{x} \mapsto F_\rho(\mathtt{y})][\mathtt{y} \mapsto F_\rho(\mathtt{x})], \mathtt{x})) \boxed{=} 2 \\ &\quad= F_\rho(\mathtt{y}) \boxed{=} 2 \end{aligned}$$

(To understand the last step, see the discussion of $\overline{access}$ below.) □

To understand how pointers are handled during the $\mathcal{WLP}$ operation, the key reinterpretations to concentrate on in $L[\mathrm{PL}]$ are the ones for the operations of the meta-language that manipulate *FVal*s (i.e., arguments of type *Val* → *Val*)—in particular, *access* and *update*. We want $\overline{access}$ and $\overline{update}$ to enjoy the following semantic properties:

$$\begin{aligned} \mathcal{T}[\![\overline{access}(FE_0, T_0)]\!]\iota &= (\mathcal{FE}[\![FE_0]\!]\iota)(\mathcal{T}[\![T_0]\!]\iota) \\ \mathcal{FE}[\![\overline{update}(FE_0, T_0, T_1)]\!]\iota &= (\mathcal{FE}[\![FE_0]\!]\iota)[\mathcal{T}[\![T_0]\!]\iota \mapsto \mathcal{T}[\![T_1]\!]\iota] \end{aligned}$$

Note that these properties require evaluating the results of $\overline{access}$ and $\overline{update}$ with respect to an arbitrary $\iota \in$ *LogicalStruct*. As mentioned earlier, it is desirable for reinterpreted base-type operations to perform simplifications whenever possible, when they construct *Term*s, *Formula*s, *FuncExpr*s, and *StructUpdate*s. However, because the value of ι is unknown, $\overline{access}$ and $\overline{update}$ operate in an uncertain environment.

$$\overline{\textit{access}}(F, k_1) \;=\; F(k_1)$$

$$\overline{\textit{access}}(\textit{FE}[k_2 \mapsto d_2]), k_1) \;=\; \begin{cases} d_2 & \text{if } (k_1 \equiv k_2) \\ \overline{\textit{access}}(\textit{FE}, k_1) & \text{if } (k_1 \neq k_2) \\ \textit{ite}(k_1 \boxed{=} k_2, d_2, \overline{\textit{access}}(\textit{FE}, k_1)) & \text{if } (k_1 \doteq k_2) \end{cases}$$

$$\overline{\textit{update}}(F, k_1, d_1) \;=\; F[k_1 \mapsto d_1]$$

$$\overline{\textit{update}}(\textit{FE}[k_2 \mapsto d_2], k_1, d_1) \;=\; \begin{cases} \textit{FE}[k_1 \mapsto d_1] & \text{if } (k_1 \equiv k_2) \\ \overline{\textit{update}}(\textit{FE}, k_1, d_1)[k_2 \mapsto d_2] & \text{if } (k_1 \neq k_2) \\ \textit{FE}[k_2 \mapsto d_2][k_1 \mapsto d_1] & \text{if } (k_1 \doteq k_2) \end{cases}$$

Figure 4.10 Simplifications performed by $\overline{\textit{access}}$ and $\overline{\textit{update}}$. The operations $\equiv$, $\neq$, and $\doteq$ denote *equality-as-terms*, *definite-disequality*, and *possible-equality*, respectively. (The possible-equality tests, "$k_1 \doteq k_2$", are really "otherwise" cases of three-pronged comparisons.)

To use semantic reinterpretation to create a $\mathcal{WLP}$ primitive that implements Morris's rule, simplifications are performed by $\overline{\textit{access}}$ and $\overline{\textit{update}}$ according to the definitions given in Fig. 4.10. The possible-equality case for $\overline{\textit{access}}$ Fig. 4.10 introduces *ite* terms. As illustrated in Ex. 4.7, it is these *ite* terms that cause the reinterpreted operations to account for possible aliasing combinations, and thus are the reason that the semantic-reinterpretation method automatically carries out the actions of Morris's rule of substitution [138].

Example 4.7 We now demonstrate how semantic reinterpretation produces the $L[\text{PL}]$ formula for $\mathcal{WLP}(*p = e, x = 5)$ claimed in Ex. 4.5.

$$\begin{aligned} U \;&:=\; \overline{\mathcal{I}}[\![*p = e]\!]U_{Id} \\ &=\; \overline{\textit{updateStore}}(U_{Id}, \overline{\mathcal{E}}[\![p]\!]U_{Id}, \overline{\mathcal{E}}[\![e]\!]U_{Id}) \\ &=\; \overline{\textit{updateStore}}(U_{Id}, \overline{\textit{lookupState}}(U_{Id}, \mathtt{p}), \overline{\textit{lookupState}}(U_{Id}, \mathtt{e}) \\ &=\; \overline{\textit{updateStore}}(U_{Id}, F_\rho(\mathtt{p}), F_\rho(\mathtt{e})) \\ &=\; ((U_{Id}\!\uparrow\!1), \{F_\rho \hookleftarrow F_\rho[F_\rho(\mathtt{p}) \mapsto F_\rho(\mathtt{e})]\}) \end{aligned}$$

$$
\begin{aligned}
&\mathcal{WLP}(*p = e, F_\rho(\mathtt{x}) \boxed{=} 5) \\
&\quad = \overline{\mathcal{F}}[\![F_\rho(\mathtt{x}) \boxed{=} 5]\!]U \\
&\quad = (\overline{\mathcal{T}}[\![F_\rho(\mathtt{x})]\!]U) \boxed{=} (\overline{\mathcal{T}}[\![5]\!]U) \\
&\quad = (\overline{\mathit{access}}(\overline{\mathcal{FE}}[\![F_\rho]\!]U, \overline{\mathcal{T}}[\![\mathtt{x}]\!]U)) \boxed{=} 5 \\
&\quad = (\overline{\mathit{access}}(\overline{\mathit{lookupFuncId}}(U, F_\rho), \overline{\mathit{lookupId}}(U, \mathtt{x}))) \boxed{=} 5 \\
&\quad = (\overline{\mathit{access}}(F_\rho[F_\rho(\mathtt{p}) \mapsto F_\rho(\mathtt{e})], \mathtt{x})) \boxed{=} 5 \\
&\quad = \mathit{ite}(F_\rho(\mathtt{p}) \boxed{=} \mathtt{x}, F_\rho(\mathtt{e}), \overline{\mathit{access}}(F_\rho, \mathtt{x})) \boxed{=} 5 \\
&\quad = \mathit{ite}(F_\rho(\mathtt{p}) \boxed{=} \mathtt{x}, F_\rho(\mathtt{e}), F_\rho(\mathtt{x})) \boxed{=} 5
\end{aligned}
$$

Note how the case for $\overline{\mathit{access}}$ that involves a possible-equality comparison causes an *ite* term to arise that tests "$F_\rho(\mathtt{p}) \boxed{=} \mathtt{x}$". The test determines whether the value of `p` is the address of `x`, which is the only aliasing condition that matters for this example. □

Although $\mathcal{WLP}$ is sometimes confused with the formula-manipulation operations used to obtain a formula that expresses it, or with the formula ψ that results, $\mathcal{WLP}$ is really a semantic notion—the set of states *described* by ψ. For example, for any statement s: *var* = *rhs*; in a language that only has `int`-valued variables, and postcondition formula φ, the formula $\varphi[\mathit{var} \leftarrow \mathit{rhs}]$ obtained by substitution is not the only formula that expresses $\mathcal{WLP}(s, \varphi)$. In fact, there are an infinity of acceptable formulas. A formula ψ is acceptable if ψ holds in the pre-state structure ι exactly when φ holds in the post-state structure $\mathcal{I}[\![s]\!]\iota$.

Definition 4.8 (Acceptable $\mathcal{WLP}$ Formula) ψ is an *acceptable* formula for $\mathcal{WLP}(s, \varphi)$ iff, for all $\iota \in$ *LogicalStruct*,

$$\mathcal{F}[\![\psi]\!]\iota = \mathcal{F}[\![\varphi]\!](\mathcal{I}[\![s]\!]\sigma),$$

where σ is the *State* that corresponds to *LogicalStruct* ι (i.e., $\sigma = ((\iota\uparrow 1), (\iota\uparrow 2)F_\rho)$; see Appendix B).

The correctness of the $\mathcal{WLP}$ primitive defined in Eqn. (4.1) is captured by the following theorem:

Theorem 4.9 For any *Stmt* s and *Formula* φ, $\psi := \overline{\mathcal{F}}[\![\varphi]\!](\overline{\mathcal{I}}[\![s]\!]U_{id})$ is an acceptable $\mathcal{WLP}$ formula for φ with respect to s.

$$
\begin{aligned}
\overline{\mathcal{U}}[\![U_3]\!]U_{1,2} &= \overline{\mathcal{U}}[\![(\emptyset, \{F'_\rho \hookleftarrow F_\rho[\mathtt{x} \mapsto F_\rho(\mathtt{x}) \boxed{\oplus} F_\rho(\mathtt{y})][\mathtt{y} \mapsto F_\rho(\mathtt{y})]\})]\!]U_{1,2} \\
&= (\emptyset, (U_{1,2}{\uparrow}2)[F_\rho \mapsto \overline{FE}[\![F_\rho[\mathtt{x} \mapsto F_\rho(\mathtt{x}) \boxed{\oplus} F_\rho(\mathtt{y})][\mathtt{y} \mapsto F_\rho(\mathtt{y})]]\!]U_{1,2}]) \\
&= (\emptyset, \{F'_\rho \hookleftarrow \overline{FE}[\![F_\rho[\mathtt{x} \mapsto F_\rho(\mathtt{x}) \boxed{\oplus} F_\rho(\mathtt{y})][\mathtt{y} \mapsto F_\rho(\mathtt{y})]]\!]U_{1,2}\}) \\
&= (\emptyset, \{F'_\rho \hookleftarrow \overline{update} \begin{pmatrix} \overline{FE}[\![F_\rho[\mathtt{x} \mapsto F_\rho(\mathtt{x}) \boxed{\oplus} F_\rho(\mathtt{y})]]\!]U_{1,2}, \\ \overline{T}[\![\mathtt{y}]\!]U_{1,2}, \\ \overline{T}[\![F_\rho(\mathtt{y})]\!]U_{1,2} \end{pmatrix}\}) \\
&= (\emptyset, \{F'_\rho \hookleftarrow (\overline{FE}[\![F_\rho[\mathtt{x} \mapsto F_\rho(\mathtt{x}) \boxed{\oplus} F_\rho(\mathtt{y})]]\!]U_{1,2})[\mathtt{y} \mapsto F_\rho(\mathtt{x})]\}) \\
&= (\emptyset, \{F'_\rho \hookleftarrow \overline{update} \begin{pmatrix} \overline{FE}[\![F_\rho]\!]U_{1,2}, \\ \overline{T}[\![\mathtt{x}]\!]U_{1,2}, \\ \overline{T}[\![F_\rho(\mathtt{x}) \boxed{\oplus} F_\rho(\mathtt{y})]\!]U_{1,2} \end{pmatrix}[\mathtt{y} \mapsto F_\rho(\mathtt{x})]\}) \\
&= (\emptyset, \{F'_\rho \hookleftarrow (F_\rho[\mathtt{x} \mapsto \overline{T}[\![F_\rho(\mathtt{x}) \boxed{\oplus} F_\rho(\mathtt{y})]\!]U_{1,2}][\mathtt{y} \mapsto F_\rho(\mathtt{x})])[\mathtt{y} \mapsto F_\rho(\mathtt{x})]\}) \\
&= (\emptyset, \{F'_\rho \hookleftarrow F_\rho[\mathtt{x} \mapsto ((F_\rho(\mathtt{x}) \boxed{\oplus} F_\rho(\mathtt{y})) \boxed{\oplus} F_\rho(\mathtt{x}))][\mathtt{y} \mapsto F_\rho(\mathtt{x})]\}) \\
&= (\emptyset, \{F'_\rho \hookleftarrow F_\rho[\mathtt{x} \mapsto F_\rho(\mathtt{y})][\mathtt{y} \mapsto F_\rho(\mathtt{x})]\}) \\
&= U_{swap}
\end{aligned}
$$

Figure 4.11 Example of symbolic composition.

Proof: See App. B.2. □

Symbolic Composition. The goal of symbolic composition is to have a method that, given two symbolic representations of state changes, computes a symbolic representation of their composed state change. In our approach, each state change is represented in logic $L[\mathrm{PL}]$ by a *StructUpdate,* and the method computes a new *StructUpdate* that represents their composition. To accomplish this, $L[\mathrm{PL}]$ is used as a reinterpretation domain, exactly as for $\mathcal{WLP}$. Moreover, $\overline{\mathcal{U}}$ *turns out to be exactly the symbolic-composition function that we seek.* In particular, $\overline{\mathcal{U}}$ works as follows:

$$
\overline{\mathcal{U}}[\![(\{I_i \hookleftarrow T_i\}, \{F_j \hookleftarrow FE_j\})]\!]U \;=\; ((U{\uparrow}1)[I_i \;\mapsto\; \overline{\mathcal{T}}[\![T_i]\!]U], (U{\uparrow}2)[F_j \;\mapsto\; \overline{\mathcal{FE}}[\![FE_j]\!]U])
$$

Example 4.10 At the syntactic level, we can demonstrate the ability of $\overline{\mathcal{U}}$ (plus simple algebraic simplification) to perform symbolic composition by showing that for the swap-code fragment from

Fig. 4.1(a)

$$\overline{\mathcal{I}}[\![s_1; s_2; s_3]\!]U_{id} = \overline{\mathcal{U}}[\![\overline{\mathcal{I}}[\![s_3]\!]U_{id}]\!](\overline{\mathcal{I}}[\![s_1; s_2]\!]U_{id}).$$

First, consider the left-hand side. As shown in Fig. 4.8, $\overline{\mathcal{I}}[\![s_1; s_2; s_3]\!]U_{id} = (\emptyset, F'_\rho \hookleftarrow F_\rho[\mathtt{x} \mapsto F_\rho(\mathtt{y})][\mathtt{y} \mapsto F_\rho(\mathtt{x})]) = U_{swap}$. Now consider the right-hand side. Let $U_{1,2}$ and U_3 be defined as follows:

$$\begin{aligned} U_{1,2} &= \overline{\mathcal{I}}[\![s_1; s_2]\!]U_{id} \\ &= (\emptyset, \{F'_\rho \hookleftarrow F_\rho[\mathtt{x} \mapsto F_\rho(\mathtt{x}) \boxed{\oplus} F_\rho(\mathtt{y})][\mathtt{y} \mapsto F_\rho(\mathtt{x})]\}) \\ U_3 &= \overline{\mathcal{I}}[\![s_3]\!]U_{id} \\ &= (\emptyset, \{F'_\rho \hookleftarrow F_\rho[\mathtt{x} \mapsto F_\rho(\mathtt{x}) \boxed{\oplus} F_\rho(\mathtt{y})][\mathtt{y} \mapsto F_\rho(\mathtt{y})]\}). \end{aligned}$$

As shown in Fig. 4.11,

$$\overline{\mathcal{U}}[\![U_3]\!]U_{1,2} = (\emptyset, \{F'_\rho \hookleftarrow F_\rho[\mathtt{x} \mapsto F_\rho(\mathtt{y})][\mathtt{y} \mapsto F_\rho(\mathtt{x})]\}).$$

Therefore, $\overline{\mathcal{U}}[\![U_3]\!]U_{1,2} = U_{swap}$. □

The semantic correctness of the symbolic-composition primitive $\overline{\mathcal{U}}$ is captured by the following theorem, which shows that the meaning of $\overline{\mathcal{U}}[\![U_2]\!]U_1$ is the composition of the meanings of U_2 and U_1:

Theorem 4.11 For all $U_1, U_2 \in$ *StructUpdate*,

$$\mathcal{U}[\![\overline{\mathcal{U}}[\![U_2]\!]U_1]\!] = \mathcal{U}[\![U_2]\!] \circ \mathcal{U}[\![U_1]\!].$$

Proof: See App. B.3. □

4.4 Symbolic Analysis for MC via Reinterpretation

To obtain the three symbolic-analysis primitives for MC, we use a reinterpretation of MC's semantics that is essentially identical to the reinterpretation for PL, modulo the fact that the semantics of PL is written in terms of the combinators *lookupEnv*, *lookupStore*, and *updateStore*, whereas the semantics of MC is written in terms of $lookup_{reg}$, $store_{reg}$, $lookup_{flag}$, $store_{flag}$, $lookup_{mem}$, and $store_{mem}$.

Symbolic Evaluation. The base types are redefined as $\overline{\mathit{BVal}} = \mathit{Formula}$, $\overline{\mathit{Val}} = \mathit{Term}$, $\overline{\mathit{State}} = \mathit{StructUpdate}$, where the vocabulary for *LogicalStruct*s is

$$(\{\texttt{zf}, \texttt{eax}, \texttt{ebx}, \texttt{ebp}, \texttt{eip}\}, \{F_{\mathit{mem}}\}).$$

Lookup and store operations for MC, such as $\overline{\mathit{lookup}}_{\mathit{mem}}$ and $\overline{\mathit{store}}_{\mathit{mem}}$, are handled the same way that $\overline{\mathit{lookupStore}}$ and $\overline{\mathit{updateStore}}$ are handled for PL.

$$\begin{aligned}
\overline{\mathit{lookup}}_{\mathit{mem}} &: \mathit{StructUpdate} \to \mathit{Term} \to \mathit{Term} \\
\overline{\mathit{lookup}}_{\mathit{mem}} &= \lambda U.\lambda T.((U{\uparrow}2)F_{\mathit{mem}})(T) \\
\overline{\mathit{store}}_{\mathit{mem}} &: \mathit{StructUpdate} \to \mathit{Term} \to \mathit{Term} \to \mathit{StructUpdate} \\
\overline{\mathit{store}}_{\mathit{mem}} &= \lambda U.\lambda T_1.\lambda T_2. \\
&\qquad ((U{\uparrow}1), (U{\uparrow}2)[F_{\mathit{mem}} \mapsto ((U{\uparrow}2)F_{\mathit{mem}})[T_1 \mapsto T_2]]) \\
\overline{\mathit{lookup}}_{\mathit{reg}} &: \mathit{StructUpdate} \to \mathit{register} \to \mathit{Term} \\
\overline{\mathit{lookup}}_{\mathit{reg}} &= \lambda U.\lambda r.(U{\uparrow}1)(r) \\
\overline{\mathit{store}}_{\mathit{reg}} &: \mathit{StructUpdate} \to \mathit{register} \to \mathit{Term} \\
&\qquad \to \mathit{StructUpdate} \\
\overline{\mathit{store}}_{\mathit{reg}} &= \lambda U.\lambda r.\lambda T.((U{\uparrow}1)[r \mapsto T], (U{\uparrow}2))
\end{aligned}$$

Because we placed `zf` in the set of constant symbols (which denote *Int32* values), we use the following definitions of $\overline{\mathit{lookup}}_{\mathit{flag}}$ and $\overline{\mathit{store}}_{\mathit{flag}}$, where in $\overline{\mathit{store}}_{\mathit{flag}}$ the *Int32* values 1 and 0 encode $\mathbb{T}$

and $\mathbb{F}$, respectively.[2]

$$\begin{aligned}
\overline{\textit{lookup}}_{\textit{flag}} &: \textit{StructUpdate} \to \textit{flagName} \to \textit{Formula} \\
\overline{\textit{lookup}}_{\textit{flag}} &= \lambda U.\lambda f.((U{\uparrow}1)(f) \boxed{=} 1) \\
\overline{\textit{store}}_{\textit{flag}} &: \textit{StructUpdate} \to \textit{flagName} \to \textit{Formula} \\
&\qquad \to \textit{StructUpdate} \\
\overline{\textit{store}}_{\textit{flag}} &= \lambda U.\lambda f.\lambda \varphi.((U{\uparrow}1)[f \mapsto \textit{ite}(\varphi, 1, 0)], (U{\uparrow}2))
\end{aligned}$$

Example 4.12 Fig. 4.1(d) shows the MC code that corresponds to the swap code in Fig. 4.1(a): lines 1–3, lines 4–6, and lines 7–9 correspond to lines 1, 2, and 3 of Fig. 4.1(a), respectively. For the MC code in Fig. 4.1(d), $\overline{\mathcal{I}}_{\text{MC}}[\![\textit{swap}]\!]U_{\textit{id}}$, which denotes the symbolic evaluation of *swap*, produces the *StructUpdate*

$$\left(\begin{array}{l}
\{\texttt{eax}' \hookleftarrow F_{\textit{mem}}(\texttt{ebp} \boxed{\texttt{-}} 14)\}, \\
\left\{\begin{array}{r}
F'_{\textit{mem}} \hookleftarrow F_{\textit{mem}}[\texttt{ebp} \boxed{\texttt{-}} 10 \mapsto F_{\textit{mem}}(\texttt{ebp} \boxed{\texttt{-}} 14)] \\
[\texttt{ebp} \boxed{\texttt{-}} 14 \mapsto F_{\textit{mem}}(\texttt{ebp} \boxed{\texttt{-}} 10)]
\end{array}\right\}
\end{array}\right)$$

Fig. 4.1(d) illustrates why it is essential to be able to handle address arithmetic: an access on a source-level variable is compiled into machine code that dereferences an address in the stack frame computed from the frame pointer (`ebp`) and an offset. This example shows that $\overline{\mathcal{I}}_{\text{MC}}$ is able to handle address arithmetic correctly. □

WLP. To create a formula for the $\mathcal{WLP}$ of φ with respect to instruction i via semantic reinterpretation, we use the reinterpreted MC semantics $\overline{\mathcal{I}}_{\text{MC}}$, together with the reinterpreted $L[\text{MC}]$ meaning function $\overline{\mathcal{F}}_{\text{MC}}$, where $\overline{\mathcal{F}}_{\text{MC}}$ is created via the same approach used in §4.3 to reinterpret $L[\text{PL}]$. $\mathcal{WLP}(i, \varphi)$ is obtained by performing $\overline{\mathcal{F}}_{\text{MC}}[\![\varphi]\!](\overline{\mathcal{I}}_{\text{MC}}[\![i]\!]U_{\textit{id}})$.

[2] To simplify the exposition, L is intentionally a limited logic over values of type *Int32*. To define $\overline{\textit{lookup}}_{\textit{flag}}$ and $\overline{\textit{store}}_{\textit{flag}}$, it would be more convenient to use a logic with Boolean-valued constant symbols $B_j \in \textit{BoolId}$, in which case a *StructUpdate* would be a triple of the form

$$(\{I_i \hookleftarrow T_i\}, \{B_j \hookleftarrow \varphi_j\}, \{F_k \hookleftarrow \textit{FE}_k\}),$$

and $\overline{\textit{lookup}}_{\textit{flag}}$ and $\overline{\textit{store}}_{\textit{flag}}$ could be defined as follows:

$$\begin{aligned}
\overline{\textit{lookup}}_{\textit{flag}} &= \lambda U.\lambda f.(U{\uparrow}2)(f) \\
\overline{\textit{store}}_{\textit{flag}} &= \lambda U.\lambda f.\lambda \varphi.((U{\uparrow}1), (U{\uparrow}2)[f \mapsto \varphi], (U{\uparrow}3))
\end{aligned}$$

```
[1] void foo(int e, int x, int* p) {
[2]     ...
[3]     *p = e;
[4]     if(x == 5)
[5]         goto ERROR;
[6] }
```

(a)

```
[1] mov  eax, p;
[2] mov  ebx, e;
[3] mov  [eax], ebx;
[4] cmp  x, 5;
[5] jz   ERROR;
[6] ...
[7] ERROR: ...
```

(b)

Figure 4.12 (a) A simple source-code fragment written in PL; (b) the MC code for (a).

Example 4.13 Fig. 4.12(a) shows a source-code fragment; Fig. 4.12(b) shows the corresponding MC code. (To simplify the MC code, source-level variable names are used.) In Fig. 4.12(a), the largest set of states just before line [3] that cause the branch to ERROR to be taken at line [4] is described by $\mathcal{WLP}(*p = e, x = 5)$. In Fig. 4.12(b), an expression that characterizes whether the branch to ERROR is taken is $\mathcal{WLP}(s_{\texttt{[1]-[5]}}, (\texttt{eip} \boxed{=} c_{\texttt{[7]}}))$, where $s_{\texttt{[1]-[5]}}$ denotes instructions [1]–[5] of Fig. 4.12(b), and $c_{\texttt{[7]}}$ is the address of ERROR. Using semantic reinterpretation,

$$\overline{\mathcal{F}}_{\text{MC}}[\![(\texttt{eip} \boxed{=} c_{\texttt{[7]}})]\!](\overline{\mathcal{I}}_{\text{MC}}[\![s_{\texttt{[1]-[5]}}]\!]U_{id})$$

produces the formula

$$(\mathit{ite}((F_{\mathit{mem}}(\texttt{p}) \boxed{=} \texttt{x}), F_{\mathit{mem}}(\texttt{e}), F_{\mathit{mem}}(\texttt{x})) \boxed{\texttt{-}} 5) \boxed{=} 0,$$

which, transliterated to informal source-level notation, is $(((p = \&x)\ ?\ e : x) - 5) = 0$.

Even though the (source-level) branch is split across two instructions in Fig. 4.12(b), $\mathcal{WLP}$ can be used to recover the branch condition. First,

$$\mathcal{WLP}(\texttt{cmp x,5; jz ERROR}, (\texttt{eip} \boxed{=} c_{\texttt{[7]}}))$$

returns the formula

$$\mathit{ite}(((F_{\mathit{mem}}(\texttt{x}) \boxed{\texttt{-}} 5) \boxed{=} 0), c_{\texttt{[7]}}, c_{\texttt{[6]}}) \boxed{=} c_{\texttt{[7]}},$$

as shown by the following derivation:

$$\begin{aligned}
\overline{\mathcal{I}}_{\text{MC}}[\![\texttt{cmp x,5}]\!]U_{id} &= (\{\texttt{zf}' \hookleftarrow \mathit{ite}((F_{\mathit{mem}}(\texttt{x}) \boxed{\text{-}} 5) \boxed{=} 0, 1, 0)\}, \emptyset) \\
&= U_1 \\
\overline{\mathcal{I}}_{\text{MC}}[\![\texttt{jz ERROR}]\!]U_1 &= \\
&\left(\left\{ \begin{array}{l} \texttt{zf}' \hookleftarrow \mathit{ite}((F_{\mathit{mem}}(\texttt{x}) \boxed{\text{-}} 5) \boxed{=} 0, 1, 0) \\ \texttt{eip}' \hookleftarrow \mathit{ite} \left(\begin{array}{l} ((F_{\mathit{mem}}(\texttt{x}) \boxed{\text{-}} 5) \boxed{=} 0), \\ c_{\texttt{[7]}}, \\ c_{\texttt{[6]}} \end{array} \right) \end{array} \right\}, \emptyset \right) \\
&= U_2
\end{aligned}$$

$$\overline{\mathcal{F}}_{\text{MC}}[\![\texttt{eip} \boxed{=} c_{\texttt{[7]}}]\!]U_2 = \mathit{ite} \left(\begin{array}{l} ((F_{\mathit{mem}}(\texttt{x}) \boxed{\text{-}} 5) \boxed{=} 0), \\ c_{\texttt{[7]}}, \\ c_{\texttt{[6]}} \end{array} \right) \boxed{=} c_{\texttt{[7]}}$$

Second, because $c_{\texttt{[7]}} \neq c_{\texttt{[6]}}$, the formula in the last line simplifies to $(F_{\mathit{mem}}(\texttt{x}) \boxed{\text{-}} 5) \boxed{=} 0$; i.e., in source-level terms, $(x - 5) = 0$. □

Symbolic Composition. For MC, symbolic composition can be performed using $\overline{\mathcal{U}}_{\text{MC}}$.

4.5 Other Language Constructs

Branching. Ex. 4.13 illustrated a $\mathcal{WLP}$ computation across a machine-code branch instruction. We now illustrate forward symbolic evaluation across a branch.

Example 4.14 Suppose that an if-statement is represented by

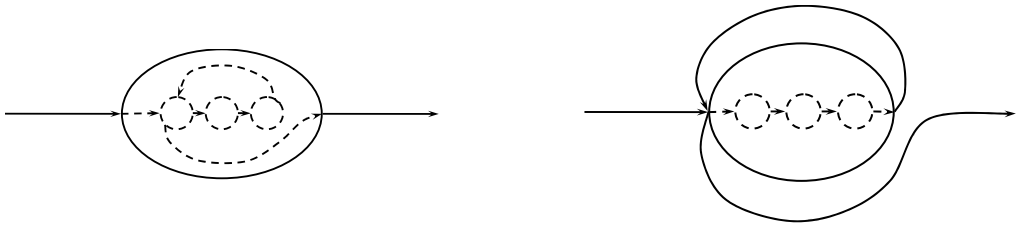
$$\mathit{IfStmt}(\mathit{BE}, \mathit{Int32}, \mathit{Int32}),$$

where *BE* is the condition and the two *Int32*s are the addresses of the true-branch and false-branch, respectively. Its factored semantics would specify how the value of the program counter PC changes:

$$\mathcal{I}[\![\mathit{IfStmt}(\mathit{BE}, c_T, c_F)]\!]\sigma = \mathit{updateStore}\ \sigma\ \text{PC}\ \mathit{cond}(\mathcal{B}[\![\mathit{BE}]\!]\sigma, \mathit{const}(c_T), \mathit{const}(c_F)).$$

Formula ObtainPathConstraintFormula(*Path* π) {
  Formula φ = $\boxed{\mathbb{T}}$; // Initial path-constraint formula
  StructUpdate $U = U_{id}$; // Initial symbolic state-transformer
  let [$\text{PC}_1 : i_1, \text{PC}_2 : i_2, \ldots, \text{PC}_n : i_n, \text{PC}_{n+1}$: skip] = π **in**
  for (k = 1; $k \leq n$; k++) {
    $U = \overline{\mathcal{I}}[\![i_k]\!]U$; // Symbolically execute i_k
    $\varphi = \varphi$ $\boxed{\&\&}$ $\overline{\mathcal{F}}[\![\text{PC}\boxed{=}\text{PC}_{k+1}]\!]U$; // Conjoin the branch condition for i_k
  }
  return φ;
}

Figure 4.13 An algorithm to obtain a path-constraint formula that characterizes which initial states must follow path π.



(a) (b)

Figure 4.14 Conversion of a recursively defined instruction—portrayed in (a) as a "microcode loop" over the actions denoted by the dashed circles and arrows—into (b), an explicit loop in the control-flow graph whose body is an instruction defined without using recursion. The three microcode operations in (b) correspond to the three operations in the body of the microcode loop in (a).

In the reinterpretation for symbolic evaluation, the *StructUpdate* U obtained by $\overline{\mathcal{I}}[\![\textit{IfStmt}(\textit{BE}, c_T, c_F)]\!]U_{id}$ would be $(\{\text{PC}' \hookleftarrow \textit{ite}(\varphi_{\textit{BE}}, c_T, c_F)\}, \emptyset)$, where $\varphi_{\textit{BE}}$ is the *Formula* obtained for *BE* under the reinterpreted semantics. To obtain the branch condition for a specific branch, say the true-branch, we evaluate $\overline{\mathcal{F}}[\![\text{PC}\boxed{=}c_T]\!]U$. The result is $(\textit{ite}(\varphi_{\textit{BE}}, c_T, c_F)\boxed{=}c_T)$, which (assuming that $c_T \neq c_F$) simplifies to $\varphi_{\textit{BE}}$. (A similar formula simplification was performed in Ex. 4.13 on the result of the $\mathcal{WLP}$ formula.)

□

Loops. One kind of intended client of our approach to creating symbolic-analysis primitives is hybrid concrete/symbolic state-space exploration [94, 167, 95, 54]. Such tools use a combination of concrete and symbolic evaluation to generate inputs that increase coverage. In such tools, a program-level loop is executed concretely a specific number of times as some path π is followed. The symbolic-evaluation primitive for a single instruction is applied to each instruction of π to obtain symbolic states at each point of π. A *path-constraint formula* that characterizes which initial states must follow π can be obtained by collecting the branch formula φ_{BE} obtained at each branch condition by the technique described above; the algorithm is shown in Fig. 4.13.

X86 String Instructions. X86 string instructions can involve actions that perform an *a priori* unbounded amount of work (e.g., the amount performed is determined by the value held in register `ecx` at the start of the instruction). This can be reduced to the loop case discussed above by giving a semantics in which the instruction itself is one of its two successors. In essence, the "microcode loop" is converted into an explicit loop (see Fig. 4.14).

Procedures. A call statement's semantics (i.e., how the state is changed by the call action) would be specified with some collection of operations. Again, the reinterpretation of the state transformer is induced by the reinterpretation of each operation:

- For a call statement in a high-level language, there would be an operation that creates a new activation record. The reinterpretation of this would generate a fresh logical constant to represent the location of the new activation record.
- For a call instruction in a machine-code language, register operations would change the stack pointer and frame pointer, and memory operations would initialize fields of the new activation record. These are reinterpreted in exactly the same way that register and memory operations are reinterpreted for other constructs.

Dynamic Allocation. Two approaches are possible:

- The allocation package is implemented as a library. One can apply our techniques to the machine code from the library.
- If a formula is desired that is based on a high-level semantics, a call statement that calls `malloc` or `new` can be reinterpreted using the kind of approach used in other systems (a fresh logical constant denoting a new location can be generated).

4.6 Incorporating Non-Determinism

Many formalisms for symbolic analysis of programs support the use of non-determinism, which is useful for writing "harness code" (code that models the possible client environments from which the code being analyzed might be called), as well as for modeling the possible inputs to a program. A common approach is to provide a primitive that returns an arbitrary value of a given type. Examples include the `SdvMakeChoice` primitive of SLAM [46] and the `havoc(x)` primitive of BoogiePL [48]. In this section, we discuss adding such a primitive, *CALL randInt32*, to MC. *CALL randInt32* is an instruction that assigns an arbitrary value to register `eax`.[3] We refer to MC extended with *CALL randInt32* as NDMC.

This section describes how implementations of the basic primitives used in symbolic program analysis are obtained for NDMC. (Essentially the same method can be applied to a version of PL extended with its own primitive for generating an arbitrary *Int32* value.)

Because our approach to creating implementations of the primitives used in symbolic program analysis is based on semantic reinterpretation, our goal is to give a concrete semantics for *CALL randInt32* whose reinterpretation produces the desired effect. At an intuitive level, we would like to treat each invocation of *CALL randInt32* as reading the next input value, and have the semantics of the program arrange to record all of the input values. To carry out something equivalent to this, we assume that the meta-language in which semantic specifications are written supports a primitive for creating a *random map*, which is a map initialized with arbitrary values.[4] Rather than recording input values, we will materialize—in a random map that is part of the input state—the

[3] In the x86 instruction set, register `eax` is used to pass back the return value from a function call.

[4] A random map is easy to model in logic L using a function that is unconstrained.

sequence of non-deterministic values that `eax` will receive on successive calls to *CALL randInt32*. The state will also contain an index-variable, which indicates the index of the next choice. Thus, all non-determinism in the concrete semantics is pushed onto the initialization of the random map in the initial state; all transitions thereafter are deterministic.

The *CALL randInt32* instruction and its semantics are defined as an extension of the MC language presented in §4.2.3:

$$\mathit{instruction} := \ldots \mid \mathit{CALL\ randInt32}$$

An NDMC state is defined in terms of

$$\mathit{choiceMap} \in \mathit{Val} \rightarrow \mathit{Val}$$
$$\mathit{choiceIndex} \in \mathit{Val}$$

and an NDMC state $\sigma \in \mathit{State}$ is now a quintuple

$$(\mathit{mem}, \mathit{reg}, \mathit{flag}, \mathit{choiceMap}, \mathit{choiceIndex}),$$

where *choiceMap* is a random map.

$$\begin{aligned}
&\mathit{lookup}_{\mathit{choiceMap}} \ : \ \mathit{State} \rightarrow \mathit{Val} \\
&\mathit{lookup}_{\mathit{choiceMap}} = \\
&\quad \lambda(\mathit{mem}, \mathit{reg}, \mathit{flag}, \mathit{choiceMap}, \mathit{choiceIndex}).\mathit{choiceMap}(\mathit{choiceIndex})
\end{aligned}$$

$$\begin{aligned}
&\mathit{incr}_{\mathit{choiceIndex}} \ : \ \mathit{State} \rightarrow \mathit{State} \\
&\mathit{incr}_{\mathit{choiceIndex}} = \\
&\quad \lambda(\mathit{mem}, \mathit{reg}, \mathit{flag}, \mathit{choiceMap}, \mathit{choiceIndex}) \\
&\qquad\qquad .(\mathit{mem}, \mathit{reg}, \mathit{flag}, \mathit{choiceMap}, \mathit{choiceIndex} + 1)
\end{aligned}$$

The concrete semantics of *CALL randInt32* is defined as follows:

$$\begin{aligned}
&\mathcal{I}[\![\mathit{CALL\ randInt32}]\!]\sigma \\
&\qquad = \mathit{incr}_{\mathtt{eip}} \left(\mathit{store}_{\mathit{reg}} \left(\begin{array}{l} \mathit{incr}_{\mathit{choiceIndex}}(\sigma), \\ \mathtt{eax}, \\ \mathit{lookup}_{\mathit{choiceMap}}(\sigma) \end{array} \right) \right)
\end{aligned}$$

Reinterpretation in Logic. As before, *State* is reinterpreted as a *StructUpdate*: $\overline{\textit{State}} = \textit{StructUpdate}$, where the vocabulary for *LogicalStruct*s is

$$\begin{pmatrix} \{\textit{choiceIndex}, \texttt{zf}, \texttt{eax}, \texttt{ebx}, \texttt{ebp}, \texttt{eip}\}, \\ \{F_{\textit{choiceMap}}, F_{\textit{mem}}\} \end{pmatrix},$$

and U_{id} is

$$\begin{pmatrix} \{\textit{choiceIndex}' \hookleftarrow \textit{choiceIndex}, \texttt{zf}' \hookleftarrow \texttt{zf}, \ldots\}, \\ \{F'_{\textit{choiceMap}} \hookleftarrow F_{\textit{choiceMap}}, F'_{\textit{mem}} \hookleftarrow F_{\textit{mem}}\} \end{pmatrix}.$$

$\mathcal{WLP}$ in the Presence of Non-Determinism. In previous sections, we have referred to the backwards-reasoning primitive generated by our method as $\mathcal{WLP}$, which is correct for the situation considered in §4.3 and 4.4, namely languages whose primitive statements/instructions are total and deterministic.

In the terminology of relational semantics [171], one considers two backwards-reasoning primitives, pre and $\widetilde{pre}$, defined as follows (where R is a binary relation on Q, and φ defines a subset of Q):

$$pre[R](\varphi) = \exists q'.\ (R(q, q') \wedge \varphi(q'))$$
$$\widetilde{pre}[R](\varphi) = \forall q'.\ (R(q, q') \Rightarrow \varphi(q'))$$

pre specifies the set of all predecessors in R of states that satisfy φ. $\widetilde{pre}$ specifies the largest set of states such that for each state q all successors of q (possibly the empty set) satisfy φ.

The backwards-reasoning primitive considered in §4.3 and 4.4 could be referred to as either pre or $\widetilde{pre}$, because the two operators are identical for total, deterministic transitions. For a non-deterministic transition system, however, pre and $\widetilde{pre}$ are different. For instance, execution of the `havoc(x)` primitive of BoogiePL [48] assigns an arbitrary value to `x`. For `havoc(x)`, pre and $\widetilde{pre}$ are defined as follows:

$$pre[\![\texttt{havoc(x)}]\!](\varphi) = \exists x.\ \varphi$$
$$\widetilde{pre}[\![\texttt{havoc(x)}]\!](\varphi) = \forall x.\ \varphi$$

The following example shows that the backwards-reasoning primitive created by our technique behaves similarly to pre.

Example 4.15 Consider what the backwards-reasoning primitive creates for $\texttt{eax} \boxed{=} 5$ with respect to *CALL randInt32*:

$$
\begin{aligned}
&\overline{\mathcal{I}}[\![\textit{CALL randInt32}]\!]U_{id} \\
&= \left(\begin{Bmatrix} \textit{choiceIndex}' \hookleftarrow (U_{id}{\uparrow}1)(\textit{choiceIndex}) \boxed{+} 1, \\ \texttt{eax}' \hookleftarrow ((U_{id}{\uparrow}2)(F_{\textit{choiceMap}}))((U_{id}{\uparrow}1)(\textit{choiceIndex})) \end{Bmatrix}, (U_{id}{\uparrow}2) \right) \\
&= \left(\begin{Bmatrix} \textit{choiceIndex}' \hookleftarrow \textit{choiceIndex} \boxed{+} 1, \\ \texttt{eax}' \hookleftarrow F_{\textit{choiceMap}}(\textit{choiceIndex}) \end{Bmatrix}, (U_{id}{\uparrow}2) \right) \\
&= U_1
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{WLP}(\textit{CALL randInt32}, \texttt{eax} \boxed{=} 5) &= \overline{\mathcal{F}}[\![\texttt{eax} \boxed{=} 5]\!]U_1 \\
&= F_{\textit{choiceMap}}(\textit{choiceIndex}) \boxed{=} 5
\end{aligned}
$$

□

$F_{\textit{choiceMap}}$ can be thought of as an array of logical variables. In the quantifier-free logic we work with, formulas are implicitly existentially quantified. Letting v denote $F_{\textit{choiceMap}}(\textit{choiceIndex})$, the formula $F_{\textit{choiceMap}}(\textit{choiceIndex}) \boxed{=} 5$ can be thought of as the quantifier-free version of the formula $\exists v.v \boxed{=} 5$, which corresponds to $pre[\![\texttt{havoc(v)}]\!](v \boxed{=} 5)$.

Thus, in earlier sections it would have been more precise to have referred to the backwards-reasoning primitive as pre, rather than $\mathcal{WLP}$—although the term $\mathcal{WLP}$ was also correct because earlier sections dealt only with languages whose primitive statements/instructions are total and deterministic.

Guaranteed Replay in the Presence of Non-Determinism. The application of directed test generation [54, 94, 95, 167] requires path constraints that enable the test-generation system to create new test inputs that are guaranteed to follow a particular path through the program.[5] In particular, during forward symbolic evaluation, we want path-constraint generation (Fig. 4.13) to

[5] See §4.7 and 4.8 for more detailed discussion of systems for directed test generation.

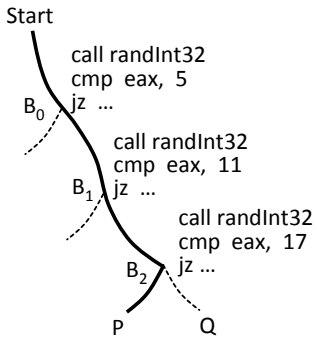


Figure 4.15 In a symbolic evaluation of the trace from *Start* to P, the three path constraints obtained from the branch instructions at B_0, B_1, and B_2 constrain the values of $F_{\mathit{choiceMap}}(0)$, $F_{\mathit{choiceMap}}(1)$, and $F_{\mathit{choiceMap}}(2)$, respectively. To create a new initial state that causes a concrete execution of the program to follow the same path, except to branch the opposite way at B_2 (to reach Q), we need the satisfying assignment returned by the theorem prover to satisfy the constraints on $F_{\mathit{choiceMap}}(0)$ and $F_{\mathit{choiceMap}}(1)$ and the negated constraint on $F_{\mathit{choiceMap}}(2)$.

produce a formula such that when a theorem prover is able to provide an assignment that satisfies the formula, the satisfying assignment serves as an initial state that will cause concrete execution of the program to follow a specific path. The paths of interest are ones that replay at least part of a previous execution trace.

The situation is illustrated in Fig. 4.15. During directed test generation, suppose that a concrete execution trace T follows the path from *Start* to P. Associated with T are three path constraints obtained from the branch instructions at B_0, B_1, and B_2. The three constraints constrain the values of $F_{\mathit{choiceMap}}(0)$, $F_{\mathit{choiceMap}}(1)$, and $F_{\mathit{choiceMap}}(2)$, respectively. To increase branch coverage, a directed-test-generation tool would like to obtain an initial state that drives the program along the same path, except when it reaches B_2, when the program should proceed to Q.

With the scheme presented in this section, the theorem prover is able to create such an initial state by providing initial values for the first three entries of $F_{\mathit{choiceMap}}$ (which models the random map *choiceMap*).

Repeatability comes from the fact that we have kept the concrete semantics deterministic by, in essence, recording all non-deterministically chosen values in a kind of shadow input stream. As a result, repeatability is automatically obtained for both symbolic evaluation as well as $\mathcal{WLP}$. In each case, for a given path we obtain an assignment for the input that forces execution along

	TSL Specifications		Generated C++ Templates	
	$\mathcal{I}[\![\cdot]\!]$	$\mathcal{F}[\![\cdot]\!] \cup \mathcal{T}[\![\cdot]\!] \cup FE[\![\cdot]\!] \cup \mathcal{U}[\![\cdot]\!]$	$\overline{\mathcal{I}}[\![\cdot]\!]$	$\overline{\mathcal{F}}[\![\cdot]\!] \cup \overline{\mathcal{T}}[\![\cdot]\!] \cup \overline{FE}[\![\cdot]\!] \cup \overline{\mathcal{U}}[\![\cdot]\!]$
x86	**3,524**	**1,510**	23,109	15,632
PowerPC	**1,546**	(already written)	12,153	15,632

Figure 4.16 The number of (non-blank) lines of C++ that are generated from the TSL specifications of the x86 and PowerPC instruction sets (as of Apr. 2010). The number of (non-blank) lines of TSL are indicated in bold.

that path: in symbolic evaluation, one works forwards and collects path constraints; in $\mathcal{WLP}$, one works backwards starting from $\mathbb{T}$; the solver is constrained to return an assignment that, at each branch instruction, causes a concrete execution to branch in the direction that stays on the path.

4.7 Implementation and Evaluation

We used TSL to (1) define the syntax of $L[\cdot]$ as a user-defined datatype; (2) create a reinterpretation based on $L[\cdot]$ formulas; (3) define the semantics of $L[\cdot]$ by writing functions that correspond to $\mathcal{T}$, $\mathcal{F}$, etc.; and (4) apply reinterpretation (2) to the meaning functions of $L[\cdot]$ itself. (We already had in hand TSL specifications of x86 and PowerPC.)

When semantic reinterpretation is performed in the manner supported by TSL, it is *independent* of any given subject language. Consequently, now that we have carried out steps (1)–(4), all three symbolic-analysis primitives can be generated automatically for a new instruction set *IS* merely by writing a TSL specification of *IS*, and then applying the TSL compiler. In essence, TSL acts as a "YACC-like" tool for generating symbolic-analysis primitives from a semantic description of an instruction set.

To illustrate the leverage gained by using the approach presented in this chapter, the table shown in Fig. 4.16 lists the number of (non-blank) lines of C++ that are generated from the TSL specifications of the x86 and PowerPC instruction sets. The number of (non-blank) lines of TSL are indicated in bold.

In addition to the components for concrete and symbolic evaluation, one also obtains an implementation of $\mathcal{WLP}$—via the method described in §4.3—by calling the C++ implementations of $\overline{\mathcal{F}}[\![\cdot]\!]$ and $\overline{\mathcal{I}}[\![\cdot]\!]$: $\mathcal{WLP}(s, \varphi) = \overline{\mathcal{F}}[\![\varphi]\!](\overline{\mathcal{I}}[\![s]\!]U_{id})$. By Thm. 4.9 of Appendix B, $\mathcal{WLP}$ is guaranteed to be consistent with the components for concrete and symbolic evaluation (modulo bugs in the implementation of TSL).

Evaluation. Some tools that use symbolic reasoning employ formula transformations that are not faithful to the actual semantics. For instance, the SAGE system for directed test generation [95] uses an approximate x86 symbolic evaluation in which concrete values are used when non-linear operators or symbolic pointer dereferences are encountered. As a result, its symbolic evaluation of a path can produce an "unfaithful" path-constraint formula φ; that is, an actual execution path may not match the program path predicted by the path-constraint formula φ. This situation is called a *divergence* [95]. Because the intended use of SAGE is to generate inputs that increase coverage, it can be acceptable for the tool to have a substantial divergence rate (due to the use of unfaithful symbolic techniques) if the cost of performing symbolic operations is lowered in most circumstances.

In contrast with directed test generation, to model check machine code [120, 174][6] an implementation of a faithful symbolic technique is required. A faithful symbolic technique could raise the cost of performing symbolic operations because faithful path-constraint formulas could be a great deal more complex than unfaithful ones. Thus, our experiment was designed to answer the question

> "What is the cost of using exact symbolic-evaluation primitives instead of unfaithful ones?"

It would have been an error-prone task to implement a faithful symbolic-evaluation primitive for x86 machine code manually. Using TSL, however, we were able to generate a faithful symbolic-evaluation primitive from an existing, well-tested TSL specification of the semantics of x86 instructions. We also generated an unfaithful symbolic-evaluation primitive that adopts SAGE's

[6] The model-checking tool for machine code is described in §5.1

approximate approach. We used these to create two directed-test-generation tools that perform state-space exploration—one that uses the faithful primitive, and one that uses the unfaithful primitive.

Although the presentation in earlier sections was couched in terms of simplified core languages, the implemented tools work with real x86 programs. Our experiment used seven C++ programs, each exercising a single algorithm from the C++ STL, compiled under Visual Studio 2005.

To compare the two tools' divergence rates and running times, we used the algorithm shown in Fig. 4.17. All execution runs were performed on a single core of a quad-core 3.0GHz Pentium Xeon processor running Windows XP, configured so that a user process has 4 GB of memory. Tab. 4.1 shows the divergence rates and running times that we measured.

Tab. 4.1 reports the number of tests executed, the average length of the trace obtained from the tests, and the average number of branches in the traces. For the faithful version, we report the average time taken for concrete execution (CE) and symbolic evaluation (SE). In the approximate ("unfaithful") version, concrete execution and symbolic evaluation were done in lock step and their total time is reported in (CE+SE). (All times are in seconds.) For each version, we also report the average time taken by the SMT solver (Yices [82]), the average number of constraints found ($|\varphi|$), and the divergence rate. For the approximate version, we also show the average distance (as a percentage of the total length of the trace) before a diverging test diverged. T_F/T_A denotes the ratio of the times (CE+SE+SMT) for the faithful version and the approximate version.

On average, the unfaithful primitive had a 57% divergence rate (computed as the arithmetic mean of the seven measured divergence rates), whereas no divergences were reported for the faithful primitive. The faithful primitive had 9.27 times more constraints in φ than the unfaithful primitive (computed as the geometric mean of the ratios of the two versions for the seven programs), and was about 1.07 times slower than the unfaithful version (geometric mean).

4.8 Related Work

Symbolic analysis is used in many recent systems for testing and verification:

```
σ := a random initial input state
Perform concrete execution, starting with input state σ, and obtain the trace T
numTracesConsidered := 0; divergences_faithful := 0; divergences_unfaithful := 0
Worklist := {⟨σ, T⟩}; AlreadyConsideredTraces := ∅
while Worklist ≠ ∅ and numTracesConsidered < threshold do
  Select and remove a pair ⟨σ, T⟩ from Worklist
  Perform two symbolic evaluations of T using the faithful and unfaithful symbolic primitives,
  respectively, generating branch predicates for each branch instruction in T
  Let B_1, B_2, ..., B_k be the branch instructions, in order, in T
  for i := k downto 1 do
    For each of the two symbolic evaluations, conjoin all the branch predicates in T prior to B_i
    with the negation of the branch predicate for B_i in T, creating path formulas φ_faithful and
    φ_unfaithful, respectively
    T_B+ := the prefix of T up to and including B_i, plus the intended successor of B_i
    if T_B+ ∈ AlreadyConsideredTraces then
      Break /* Exit the for loop; all prefixes of T_B+ are in AlreadyConsideredTraces, too */
    else
      Insert T_B+ into AlreadyConsideredTraces
    end if
    if φ_faithful is unsatisfiable then
      Continue /* Go to the next iteration of the for loop */
    end if
    σ'_faithful := a satisfying assignment for φ_faithful
    Perform concrete execution, starting with input state σ'_faithful, and obtain the trace T'
    numTracesConsidered := numTracesConsidered + 1
    if T' does not match T_B+ then
      Increment divergences_faithful by 1
    end if
    if φ_unfaithful is unsatisfiable then
      Increment divergences_unfaithful by 1
    else
      σ'_unfaithful := a satisfying assignment for φ_unfaithful
      Perform concrete execution, starting with input state σ'_faithful, and obtain the trace T''
      if T'' does not match T_B+ then
        Increment divergences_unfaithful by 1
      end if
    end if
    Insert ⟨σ'_faithful, T'⟩ into Worklist
  end for
end while
```

Figure 4.17 Directed-test-generation algorithm used for comparing the divergence rates of the faithful and unfaithful symbolic-evaluation primitives.

Name (STL)	#Tests	|Trace| (#Instrs)	#Branches	Faithful					Approximate					Slowdown
				CE	SE	SMT	$\|\varphi\|$	Div.	CE+SE	SMT	$\|\varphi\|$	Div.	Dist.	(T_F/T_A)
copy	12	1462	19	0.3	3.44	0.017	**6**	**0%**	3.58	0.013	**1**	**50%**	93%	1.05
equal	202	1604	64	0.33	5.56	0.48	**54**	**0%**	5.75	0.46	**24**	**60%**	73%	1.11
find	344	1240	174	0.15	5.34	0.2	**144**	**0%**	5.31	0.17	**85**	**50%**	82%	1.07
partition	19	1293	43	0.24	5.26	0.79	**43**	**0%**	5.43	0.26	**1**	**73%**	87%	1.16
random_shuffle	94	2448	71	0.48	7.56	0.028	**37**	**0%**	7.88	0.014	**1**	**48%**	99%	1.03
search	274	1422	107	0.33	6.3	0.17	**59**	**0%**	6.37	0.13	**31**	**55%**	89%	1.07
transform	200	3749	95	0.82	18.56	0.05	**85**	**0%**	19.36	0.012	**1**	**64%**	99%	1.00

Table 4.1 Experimental results. Key: CE = time for concrete execution; SE = time for symbolic execution; SMT = solver time; $|\varphi|$ = avg. number of constraints found; Div. = divergence rate; CD+SE = time for concrete + symbolic execution (when run in lock-step); Dist. = avg. distance before a diverging test diverges. T_F/T_A denotes the ratio of the times (CE+SE+SMT) for the faithful version and the approximate version. (All times are in seconds.)

- Hybrid concrete/symbolic tools for directed test generation [54, 94, 95, 167] use a combination of concrete and symbolic evaluation to generate inputs that increase coverage. They use concrete evaluation to identify an executable path π. They use symbolic evaluation to obtain a path formula for π, then change the formula to be one for a path π' that follows the same sequence of branches as π, except that at the final branch node π' branches in the direction opposite to the one taken by π, and call an SMT solver to determine if there is an input that drives the program down π'.
- $\mathcal{WLP}$ can be used to create new predicates that split part of a program's abstract state space [46, 49].
- Symbolic composition is useful when a tool has access to a formula that summarizes a called procedure's behavior [186]; re-exploration of the procedure is avoided by symbolically composing a path formula with the procedure-summary formula.

However, compared with the way such symbolic-analysis primitives are implemented in existing program-analysis tools, our work has one definite advantage: it creates the key concrete-execution and symbolic-analysis components in a way that ensures by construction that they are *mutually consistent*.

We use a *declarative approach*: one provides a specification of the subject language's *standard* semantics; then, as described in §4.3 and 4.4, mutually-consistent implementations of symbolic evaluation, $\mathcal{WLP}$, and symbolic composition are obtained from the subject language's standard semantics by (i) reinterpreting meta-language constructs in terms of logic, and (ii) reinterpreting a logic's meaning functions. The advantage of this approach is that one obtains implementations of (a) concrete execution, (b) symbolic evaluation, (c) $\mathcal{WLP}$, and (d) symbolic composition from a *single* specification, which removes the possibility of different analysis components having different "views" of the semantics.

It appears to be the case that in most tools, the concrete-execution and symbolic-analysis primitives are not implemented in a way that guarantees such a consistency property. For instance, in the source code for B2 [106] (the next-generation BLAST), one finds symbolic evaluation ($post$) and $\mathcal{WLP}$ implemented with different pieces of code, and hence mutual consistency is not guaranteed. $\mathcal{WLP}$ is implemented via substitution, with special-case code for handling pointers. Any modification of the B2 intermediate representation would require changing both $post$ and $\mathcal{WLP}$, and possibly rethinking the substitution method.

Recently, directed-test-generation tools have been created for x86 executables—e.g., SAGE [95] and BITSCOPE [54].

- BITSCOPE is a framework that takes an x86 executable and provides information about execution paths that can be used for additional, more specific analyses, such as finding out what inputs cause erroneous behavior. To perform symbolic evaluation, they first translate each x86 instruction into an intermediate representation that is designed to model the semantics of the original x86 instruction, including all implicit side effects (such as flags that are set), register addressing modes, and other issues. Symbolic evaluation is performed on the IR with a symbolic transformer for each IR statement.
- SAGE is a *white-box fuzz-testing tool* for x86 Windows applications [95]. The system uses *offline, trace-based* constraint generation: concrete execution and symbolic evaluation are performed over a separately recorded, replayable execution trace in which the outcome of each nondeterministic event encountered during the recorded run has been captured. To

generate path constraints, SAGE maintains a concrete state and a symbolic state—a pair of stores that associate each memory location and register to a byte-sized value and a *symbolic tag*, which is an expression that represents either an input value or a function of some input values. A symbolic tag is propagated on the trace during the process of symbolic evaluation by using a symbolic transformer written specifically for each instruction. The concrete store is sometimes used to *concretize* symbolic values that are overly complex. In SAGE, symbolic pointer dereferences are intentionally ignored to reduce complexity. SAGE could be improved to increase coverage by using more precise path constraints created from the symbolic-evaluation primitive produced by our technique. §4.7 shows that the faithful constraints created by our technique dramatically reduce the number of divergences with only a modest (7%) increase in running time.

BITSCOPE uses the approach of translating each instruction to a common intermediate representation (CIR) (see §4.1), which provides a level of assurance that the concrete-execution and symbolic-evaluation components are mutually consistent. SAGE uses independently created components for capturing execution traces and for path-constraint generation. It also uses approximate techniques during the symbolic-evaluation part of constraint generation; hence, the treatment of program semantics in SAGE is definitely inconsistent, which causes divergences. ($\mathcal{WLP}$ and symbolic composition do not play a role in either SAGE or BITSCOPE.)

Relationship to Partial Evaluation, Binding-Time Analysis, and 2-Level Semantics. In general, the semantic definition of an imperative programming language is a meaning function $\mathcal{I}$ with type $\mathcal{I} : \mathit{Stmt} \times \mathit{State} \rightarrow \mathit{State}$. The objective of a primitive for symbolic evaluation can be stated as follows:

> Given the semantic definition of a programming language, $\mathcal{I} : \mathit{Stmt} \times \mathit{State} \rightarrow \mathit{State}$, together with a specific programming-language statement (or instruction) $s \in \mathit{Stmt}$, create a logical formula that captures the semantics of s.

Given such a goal for the primitive to be created, it is not surprising that partial-evaluation techniques come into play in the tool that generates implementations of such primitives. In essence, we

wish to partially evaluate $\mathcal{I}$ with respect to *Stmt* s so that the residual object captures the semantics of s, while at the same time the result is translated to L. Semantic reinterpretation permits us to do this: let U_s be the *StructUpdate* $\overline{\mathcal{I}}[\![s]\!]U_{id}$. Then U_s is the partial evaluation of $\mathcal{I}$ with respect to s, translated to logic.

In our implementation, the TSL system is supplied with a TSL program for the meaning function $\mathcal{I}$ (i.e., interpInstr). Although TSL is not a partial-evaluation system *per se*, for reasons discussed in §3.2.1, the TSL compiler performs binding-time analysis [108], and annotates the code for interpInstr to create an intermediate representation in a two-level language [149]. In our case, Level 1 corresponds to parameter `I` of interpInstr, and Level 2 corresponds to parameter `state`. To generate implementations of symbolic-analysis primitives via semantic reinterpretation, we use two different reinterpretations for the two levels:

- Concrete semantics (C) for Level 1.
- Something close to the Herbrand interpretation (H) for Level 2: operators of L are used as syntactic constructors, but algebraic simplifications are performed whenever possible.

Let interpInstr-`CH` denote interpInstr-`2level` reinterpreted in this fashion. When interpInstr-`CH` is executed, it creates a residual expression as output. Because concrete semantics is used for level 1, all parts of interpInstr that are not relevant to the form of `I` are eliminated.

Overall, the TSL compiler and the two interpretations create something that is very similar to a generating extension [108] interpInstr-`gen` for interpInstr. If `p` is a two-input program, a *generating extension* `p-gen` is any program with the property that for every input pair a and b,

$$[\![\texttt{p-gen}]\!](a) = \texttt{p}_a, \text{ where } [\![\texttt{p}_a]\!](b) = [\![\texttt{p}]\!](a, b).$$

Thus, $\mathcal{I}$-`gen` is a program such that for every statement s and *State* σ,

$$[\![\mathcal{I}\text{-}\texttt{gen}]\!](s) = \mathcal{I}_s, \text{ where } [\![\mathcal{I}_s]\!](\sigma) = [\![\mathcal{I}]\!](s, \sigma).$$

Generating extension interpInstr-`gen` would be a program with the following property:

$$\begin{aligned} [\![\text{interpInstr-}\texttt{gen}]\!](\texttt{I}) &= \text{interpInstr}_{\texttt{I}}, \text{where} \\ [\![\text{interpInstr}_{\texttt{I}}]\!](\texttt{S}) &= [\![\text{interpInstr}]\!](\texttt{I}, \texttt{S}). \end{aligned}$$

interpInstr-CH has similar properties:

$$\begin{aligned} [\![\text{interpInstr-CH}]\!](\mathtt{I}, U_{id}) &= U_{\mathtt{I}}, \text{where} \\ \mathcal{U}[\![U_{\mathtt{I}}]\!](\mathtt{S}) &= [\![\text{interpInstr}]\!](\mathtt{I}, \mathtt{S}). \end{aligned}$$

Consequently, interpInstr-gen and interpInstr-CH are not the same, although the difference between is quite small. interpInstr-CH still requires *two* inputs to be supplied (but we could use the trivial value U_{id} for the second input).

When partial-evaluation machinery is included in the discussion, the explanation is complicated by the number of language levels involved. Consequently, in this chapter we chose to base the discussion on the simpler principle of semantic reinterpretation, which has benefits and drawbacks:

- The benefit is that the explanation is simpler, and could also be useful for direct hand implementation when a meta-system such as TSL is not available.
- The drawback is that in some of the sections it may appear that many steps perform rather trivial transliteration of expressions from programming language PL_i into expressions of the corresponding logic $L[\text{PL}_i]$. In part, this is an artifact of trying to present the method in an easy-to-digest manner; in part, it mimics the behavior of a generating extension: copying (or transliterating) the appropriate residual expression is one of the principles of "writing a generating extension by hand" [51, 123].

4.9 Conclusion

This chapter presents a way to obtain automatically mutually-consistent, correct-by-construction implementations of symbolic primitives—in particular, quantifier-free, first-order-logic formulas for (a) symbolic evaluation of a single command, (b) $\mathcal{WLP}$ with respect to a single command, and (c) symbolic composition for a class of formulas that express state transformations. The approach presented in the chapter involves *generating* implementations of each of the primitives from a single specification of the subject language's concrete semantics. The generated implementations are guaranteed to be mutually consistent (modulo bugs in the implementation of

the program-generation implementation), and also to be consistent with an instruction-set emulator (for concrete execution) that is generated from the same specification of the subject language's concrete semantics.

In this work, the method used to generate such implementations is semantic reinterpretation, a technique originally introduced by Mycroft and Jones [110, 144] as a method for formulating abstract interpretations. In this work, we are not doing abstract interpretation *per se* (i.e., to over-approximate the concrete semantics [73]), but we take two-fold advantage of their methodology: we use two separate semantic reinterpretations—(i) reinterpretation of a *programming language's* meaning function(s), and (ii) reinterpretation of a *logic's* meaning function(s). The two kinds of reinterpretations define the key primitives $\overline{\mathcal{I}}$, $\overline{\mathcal{F}}$, and $\overline{\mathcal{U}}$ from which the desired implementations of symbolic evaluation, $\mathcal{WLP}$, and symbolic composition are obtained.

As far as we are aware, the application of semantic reinterpretation to a logic is a new idea. A related innovation on which our results rest was to define a particular form of state-transformation formula (structure-update expressions) as a first-class notion in the logic. By this device, such formulas could (i) serve as a replacement domain in the reinterpretations of both the programming language's meaning functions and the logic's meaning functions, and (ii) be reinterpreted themselves.

We applied our technique to both the x86 and PowerPC instruction sets, using the TSL system as our implementation platform. §4.7 discusses the substantial leverage that we obtained using TSL's facilities for semantic reinterpretation: from 6,580 lines of TSL, 101,788 lines of C++ were produced that implement $\mathcal{I}$, $\overline{\mathcal{I}}$, $\overline{\mathcal{F}}$, $\overline{\mathcal{T}}$, $\overline{FE}$, and $\overline{\mathcal{U}}$ for x86 and PowerPC. Moreover, for each instruction set all six primitives are guaranteed to be mutually consistent (modulo bugs in the implementation of TSL and in the implementations of the primitives for the two kinds of reinterpretations).

As proposed by Mycroft and Jones [110, 144], in a semantic reinterpretation one refactors the specification of a language's concrete semantics into a suitable form by introducing appropriate combinators that are subsequently redefined. While this style of semantic reinterpretation is supported by the TSL system, ordinarily one never has to be concerned with refactoring a specification. Instead, each reinterpretation is performed at the meta-level; that is, each reinterpretation involves

redefining the approximately 40 primitives of the TSL meta-language.[7] In our TSL-based semantic reinterpretations of specifications of the concrete semantics of x86 and PowerPC, we did not have to refactor the specification to introduce any special combinators.

Finally, we conducted an experiment that used the generated primitives on x86 code, compiled under Visual Studio 2005 from C++ STL source code, to gain insight on the question "What is the cost of using exact symbolic-evaluation primitives instead of unfaithful ones in a system for directed test generation?" The experiment showed that using exact symbolic-analysis primitives, as opposed to ones that approximate the real semantics, is slower by a factor of 1.07, but is dramatically more accurate.

[7] Each of the numeric primitives comes in four bit-widths: 8-bit, 16-bit, 32-bit, and 64-bit. All four must be reinterpreted; however, generally the reinterpretation of a given family of four such numeric primitives can be parameterized on bit-width, so we only count each family as a single primitive.

Chapter 5

Case Studies

This chapter discusses two applications that use the TSL-generated analysis components. Both applications use logic-based search procedures to establish properties of machine-code programs. Compared to work by others on logic-based search procedures for machine code, what distinguishes the work described in this chapter is that both applications are *goal-directed*. That is, they both have a target property or program point of interest, and this target is used to focus the search. More discussion of related work is found in §5.1.5 and §5.2.9.

§5.1 presents the algorithms used in MCVETO (**M**achine-**C**ode **VE**rification **TO**ol), a tool to check whether a stripped machine-code program satisfies a safety property. The verification problem that MCVETO addresses is challenging because it cannot assume that it has access to (i) certain structures commonly relied on by source-code verification tools, such as control-flow graphs and call-graphs, and (ii) meta-data, such as information about variables, types, and aliasing. It cannot even rely on out-of-scope local variables and return addresses being protected from the program's actions. What distinguishes MCVETO from other work on software model checking is that it shows how verification of machine code can be performed, while avoiding conventional techniques that would be unsound if applied at the machine-code level.

Botnets are a major threat to the security of computer systems and the Internet. An increasing number of individual Internet sites have been compromised by attacks from across the world to become part of various kinds of malicious botnets. §5.2 presents a tool, called BCE, for automatically extracting botnet-command information from bot executables. BCE helps analyzing the behavior of bots by providing proper input commands that trigger malicious behaviors.

Both applications make use of TSL-generated analysis components, including concrete execution as well as the symbolic-analysis primitives presented in Chapter 4. MCVETO also uses several TSL-generated static-analysis components, including ARA (§3.3.2) and ASI (§3.3.4).

5.1 MCVETO

As discussed in Chapter 2, machine-code analysis presents many new challenges. For instance, at the machine-code level, memory is one large byte-addressable array, and an analyzer must handle computed—and possibly non-aligned—addresses. It is crucial to track array accesses and updates accurately; however, the task is complicated by the fact that arithmetic and dereferencing operations are both pervasive and inextricably intermingled. For instance, if local variable `x` is at offset –12 from the activation record's frame pointer (register `ebp`), an access on `x` would be turned into an operand [`ebp`–12]. Evaluating the operand first involves pointer arithmetic ("`ebp`–12") and then dereferencing the computed address ("[·]"). On the other hand, machine-code analysis also offers new opportunities, in particular, the opportunity to track low-level, platform-specific details, such as memory-layout effects. Programmers are typically unaware of such details; however, they are often the source of exploitable security vulnerabilities.

The algorithms used in software model checkers that work on source code [47, 49, 102] would be be unsound if applied to machine code. For instance, before starting the verification process proper, SLAM [47] and BLAST [102] perform flow-insensitive (and possibly field-sensitive) points-to analysis. However, such analyses often make unsound assumptions, such as assuming that the result of an arithmetic operation on a pointer always remains inside the pointer's original target. Such an approach assumes—without checking—that the program is ANSI C compliant, and hence causes the model checker to ignore behaviors that are allowed by some compilers (e.g., arithmetic is performed on pointers that are subsequently used for indirect function calls; pointers move off the ends of structs or arrays, and are subsequently dereferenced). A program can use such features for good reasons—e.g., as a way for a C program to simulate subclassing [172]—but they can also be a source of bugs and security vulnerabilities.

In this work, we developed a model checker for machine code, called MCVETO (**M**achine-**C**ode **VE**rification **TO**ol).[1] MCVETO uses *directed proof generation* (DPG) [98] to find either an input that causes a (bad) target state to be reached, or a proof that the bad state cannot be reached. (The third possibility is that MCVETO fails to terminate.)

What distinguishes the work on MCVETO is that it addresses a large number of issues that have been ignored in previous work on software model checking, and would cause previous techniques to be unsound if applied to machine code. The contributions of our work can be summarized as follows:

1. We show how to verify safety properties of machine code while avoiding a host of assumptions that are unsound in general, and that would be inappropriate in the machine-code context, such as reliance on symbol-table, debugging, or type information, and preprocessing steps for (a) building a precomputed, fixed, interprocedural control-flow graph (ICFG), or (b) performing points-to/alias analysis.
2. MCVETO builds its (sound) abstraction of the program's state space on-the-fly, performing disassembly one instruction at a time during state-space exploration, without static knowledge of the split between code vs. data. (It does not have to be prepared to disassemble *collections* of nested branches, loops, procedures, or the whole program all at once, which is what can confuse conventional disassemblers [128].)

   The initial abstraction has only two abstract states, defined by the predicates "PC $=$ *target*" and "PC $\neq$ *target*" (where "PC" denotes the program counter). The abstraction is gradually refined as more of the program is exercised (§5.1.2). MCVETO can analyze programs with instruction aliasing[2] because it builds its abstraction of the program's state space entirely on-the-fly. Moreover, MCVETO is capable of verifying (or detecting flaws in) self-modifying code (SMC). With SMC there is no fixed association between an address and the instruction

[1] MCVETO was carried out in collaboration primarily with A. Thakur, A. Lal, and T. Reps, along with A. Burton, D. Driscoll, M. Elder, and T. Andersen. My contribution to the work consisted of the TSL-generated anaysis components for concrete execution and symbolic execution, discussed in Chapter 4, along with the development of the techniques described in §5.1.2.1 and §5.1.2.2.

[2] Programs written in instruction sets with varying-length instructions, such as x86, can have "hidden" instructions starting at positions that are out of registration with the instruction boundaries of a given reading of an instruction stream [128].

at that address, but this is handled automatically by MCVETO's mechanisms for abstraction refinement. To the best of our knowledge, MCVETO is the first model checker to handle SMC.

3. MCVETO introduces *trace generalization*, a new technique for eliminating *families* of infeasible traces. Compared to prior techniques that also have this ability [50, 101], our technique involves *no calls on an SMT solver*, and *avoids the potentially expensive step of automaton complementation*.
4. MCVETO introduces a new approach to performing DPG (Directed Proof Generation) on multi-procedure programs. Godefroid et al. [96] presented a declarative framework that codifies the mechanisms used for DPG in SYNERGY [98], DASH [49], and SMASH [96] (which are all instances of the framework). In their framework, *inter*procedural DPG is performed by invoking *intra*procedural DPG as a subroutine. In contrast, MCVETO's algorithm lies outside of that framework: the interprocedural component of MCVETO uses (and refines) an *infinite graph*, which is finitely represented and queried by *symbolic operations*.
5. We developed a language-independent algorithm to identify the aliasing condition relevant to a property in a given state (§5.1.2.1). Unlike previous techniques [49], it applies when static names for variables/objects are unavailable.
6. We developed several techniques to enhance the methods used during DPG to elaborate the abstraction in use. Although these techniques are speculative, *soundness is retained* at all times.

Items 1 and 2 address execution details that are typically ignored (unsoundly) by source-code analyzers. Item 2 is specific to machine-code analysis. 3, 4, 5, and 6 are applicable to both source-code and machine-code analysis. MCVETO is not restricted to an impoverished language. In particular, it handles pointers and bit-vector arithmetic.

We implemented MCVETO in a language-independent way by using the TSL system to implement the analysis components needed by MCVETO—i.e., (a) an emulator for running tests, (b) a primitive for performing symbolic execution, and (c) a primitive for the pre-image operator (Pre). In addition, we developed language-independent approaches to the issues discussed above

(e.g., item 5). As discussed in Chapter 3, the TSL system acts as a "YACC-like" tool for creating versions of MCVETO for different instruction sets: given an instruction-set description, a version of MCVETO is generated automatically. We created two such instantiations of MCVETO from descriptions of the Intel x86 and PowerPC instruction sets.

The remainder of this section is organized as follows: §5.1.1 contains a brief review of DPG. §5.1.2 explains the methods used to achieve the contributions of MCVETO. §5.1.3 describes how different instances of MCVETO are generated automatically by using the TSL system. §5.1.4 presents experimental results. §5.1.5 discusses related work. §5.1.6 concludes.

5.1.1 Background on Directed Proof Generation (DPG)

Given a program P and a particular control location *target* in P, DPG returns either an input for which execution leads to *target* or a proof that *target* is unreachable (or DPG does not terminate). Two approximations of P's state space are maintained:

- A set T of concrete traces, obtained by running P with specific inputs. T *under*approximates P's state space.
- A graph G, called the *abstract graph*, obtained from P via abstraction (and abstraction refinement). G *over*approximates P's state space.

Nodes in G are labeled with formulas; edges are labeled with program statements or program conditions. One node is the *start node* (where execution begins); another node is the *target node* (the goal to reach). Information to relate the under- and overapproximations is also maintained: a concrete state σ in a trace in T is called a *witness* for a node n in G if σ satisfies the formula that labels n.

If G has no path from *start* to *target*, then DPG has proved that *target* is unreachable, and G serves as the proof. Otherwise, DPG locates a *frontier*: a triple (n, I, m), where (n, m) is an edge on a path from *start* to *target* such that n has a witness w but m does not, and I is the instruction on (n, m). DPG either performs concrete execution (attempting to reach *target*) or refines G by splitting nodes and removing certain edges (which may prove that *target* is unreachable). Which action to perform is determined using the basic step from *directed test generation* [94], which uses

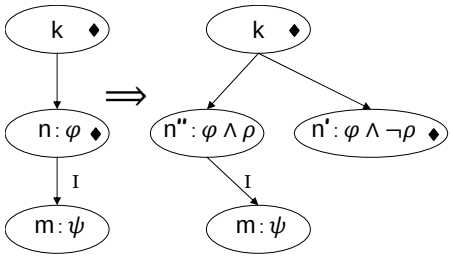


Figure 5.1 The general refinement step across frontier (n, I, m). The presence of a witness is indicated by a "♦" inside of a node.

symbolic execution to try to find an input that allows execution to cross frontier (n, I, m). Symbolic execution is performed over symbolic states, which have two components: a *path constraint,* which represents a constraint on the input state, and a *symbolic map*, which represents the current state in terms of input-state quantities. DPG performs symbolic execution along the path taken during the concrete execution that produced witness w for n; it then symbolically executes I, and conjoins to the path constraint the formula obtained by evaluating m's predicate ψ with respect to the symbolic map. It calls an SMT solver to determine if the path constraint obtained in this way is satisfiable. If so, the result is a satisfying assignment that is used to add a new execution trace to T. If not, DPG refines G by splitting node n into n' and n'', as shown in Fig. 5.1.

Refinement changes G to represent some *non-connectivity* information: in particular, n' is not connected to m in the refined graph (see Fig. 5.1). Let ψ be the formula that labels m, c be the concrete witness of n, and S_n be the symbolic state obtained from the symbolic execution up to n. DPG chooses a formula ρ, called the *refinement predicate,* and splits node n into n' and n'' to distinguish the cases when n is reached with a concrete state that satisfies ρ (n'') and when it is reached with a state that satisfies $\neg\rho$ (n'). The predicate ρ is chosen such that (i) no state that satisfies $\neg\rho$ can lead to a state that satisfies ψ after the execution of I, and (ii) the symbolic state S_n satisfies $\neg\rho$. Condition (i) ensures that the edge from n' to m can be removed. Condition (ii) prohibits extending the current path along I (forcing the DPG search to explore different paths). It also ensures that c is a witness for n' and not for n'' (because c satisfies S_n)—and thus the frontier during the next iteration must be different.

5.1.2 MCVETO

In this section, we focus on explaining the language-independent algorithm that we developed to identify the aliasing condition relevant to a property in a given state (§5.1.2.1), and the mechanisms to discover candidate invariants from a trace, which are then incorporated into the abstract graph (§5.1.2.2). The details of contributions 1, 2, 3, and 4 listed in the introduction to §5.1 can be found in the full paper ([174, 175]).

5.1.2.1 A Language-Independent Approach to Aliasing Relevant to a Property

This section describes how MCVETO identifies—in a language-independent way suitable for use with machine code—the aliasing condition relevant to a property in a given state (contribution 5 from the introduction to §5.1). Chapter 4 showed how to generate a pre-image primitive Pre for machine code; however, repeated application of Pre causes refinement predicates to explode. We now present a language-independent algorithm for obtaining an aliasing condition α that is suitable for use in machine-code analysis. From α, one immediately obtains Pre_α. There are two challenges to defining an appropriate notion of aliasing condition for use with machine code: (i) `int`-valued and address-valued quantities are indistinguishable at runtime, and (ii) arithmetic on addresses is used extensively.

Suppose that the frontier is (n, I, m), ψ is the formula on m, and S_n is the symbolic state obtained via symbolic execution of a concrete trace that reaches n. For source code, Beckman et al. [49] identify aliasing condition α by looking at the relationship, in S_n, between the addresses written to by I and the ones used in ψ. However, their algorithm for computing α is language-*dependent*: their algorithm has the semantics of C implicitly encoded in its search for "the addresses written to by I". In contrast, as explained below, we developed an alternative, language-*independent* approach, both to identifying α and computing Pre_α.

For the moment, to simplify the discussion, suppose that a concrete machine-code state is represented using two maps M : *INT* $\rightarrow$ *INT* and R : *REG* $\rightarrow$ *INT*. Map M represents memory, and map R represents the values of machine registers. (A more realistic definition of memory is considered later in this section.) We use the standard theory of arrays to describe

(functional) updates and accesses on maps, e.g., *update*(m, k, d) denotes the map m with index k updated with the value d, and *access*(m, k) is the value stored at index k in m. (We use the notation $m(r)$ as a shorthand for *access*(m, r).) We also use the standard axiom from the theory of arrays: $(\textit{update}(m, k_1, d))(k_2) = \textit{ite}(k_1 = k_2, d, m(k_2))$, where *ite* is an *if-then-else* term. Suppose that I is "`mov [eax],5`" (which corresponds to `*eax = 5` in source-code notation) and that ψ is $(M(R(\texttt{ebp}) - 8) + M(R(\texttt{ebp}) - 12) = 10)$.[3] First, we symbolically execute I starting from the identity symbolic state $S_{id} = [M \mapsto M, R \mapsto R]$ to obtain the symbolic state $S' = [M \mapsto \textit{update}(M, R(\texttt{eax}), 5), R \mapsto R]$. Next, we evaluate ψ under S'—i.e., perform the substitution $\psi[M \leftarrow S'(M), R \leftarrow S'(R)]$. For instance, the term $M(R(\texttt{ebp}) - 8)$, which denotes the contents of memory at address $R(\texttt{ebp}) - 8$, evaluates to $(\textit{update}(M, R(\texttt{eax}), 5))(R(\texttt{ebp}) - 8)$. From the axiom for arrays, this simplifies to $\textit{ite}(R(\texttt{eax}) = R(\texttt{ebp}) - 8, 5, M(R(\texttt{ebp}) - 8))$. Thus, the evaluation of ψ under S' yields

$$\begin{pmatrix} \textit{ite}(R(\texttt{eax}) = R(\texttt{ebp}) - 8, 5, M(R(\texttt{ebp}) - 8)) \\ + \ \textit{ite}(R(\texttt{eax}) = R(\texttt{ebp}) - 12, 5, M(R(\texttt{ebp}) - 12)) \end{pmatrix} = 10 \tag{5.1}$$

This formula equals $\text{Pre}(I, \psi)$ as discussed in [125] and Chapter 4.

The process described above illustrates a general property: for any instruction I and formula ψ, $\text{Pre}(I, \psi) = \psi[M \leftarrow S'(M), R \leftarrow S'(R)]$, where $S' = \text{SE}[\![I]\!]S_{id}$ and $\text{SE}[\![\cdot]\!]$ denotes symbolic execution [125].

The next steps are to identify α and to create a simplified formula ψ' that weakens $\text{Pre}(I, \psi)$. These are carried out simultaneously during a traversal of $\text{Pre}(I, \psi)$ that makes use of the symbolic state S_n at node n. We illustrate this on the example discussed above for a case in which $S_n(R) = [\texttt{eax} \mapsto R(\texttt{ebp}) - 8]$ (i.e., continuing the scenario from footnote 3, `eax` holds `&x`). Because the *ite*-terms in Eqn. (5.1) were generated from array accesses, *ite*-conditions represent possible constituents of aliasing conditions. We initialize α to *true* and traverse Eqn. (5.1). For each subterm t of the form $\textit{ite}(\varphi, t_1, t_2)$ where φ definitely holds in symbolic state S_n, t is simplified to t_1 and φ is conjoined to α. If φ can never hold in S_n, t is simplified to t_2 and $\neg\varphi$ is conjoined to α. If φ can sometimes hold and sometimes fail to hold in S_n, t and α are left unchanged.

[3] In x86, `ebp` is the frame pointer, so if program variable `x` is at offset –8 and `y` is at offset –12, ψ corresponds to $\texttt{x} + \texttt{y} = 10$.

In our example, $R(\texttt{eax})$ equals $R(\texttt{ebp}) - 8$ in symbolic state S_n; hence, applying the process described above to Eqn. (5.1) yields

$$\begin{aligned} \psi' &= (5 + M(R(\texttt{ebp}) - 12) = 10) \\ \alpha &= (R(\texttt{eax}) = R(\texttt{ebp}) - 8) \wedge (R(\texttt{eax}) \neq R(\texttt{ebp}) - 12) \end{aligned} \tag{5.2}$$

The formula $\alpha \Rightarrow \psi'$ is the desired refinement predicate $\text{Pre}_\alpha(I, \psi)$.

In practice, we found it beneficial to use an alternative approach, which is to perform the same process of evaluating conditions of *ite* terms in $\text{Pre}(I, \psi)$, but to use one of the concrete witness states W_n of frontier node n in place of symbolic state S_n. The latter method is less expensive (it uses formula-evaluation steps in place of SMT solver calls), but generates an aliasing condition specific to W_n rather than one that covers all concrete states described by S_n.

Both approaches are *language-independent* because they isolate where the instruction-set semantics comes into play in $\text{Pre}(I, \psi)$ to the computation of $S' = \text{SE}[\![I]\!]S_{id}$; all remaining steps involve only purely logical primitives.[4] Although our algorithm computes $\text{Pre}(I, \psi)$ explicitly, that step alone does not cause an explosion in formula size; explosion is due to *repeated* application of Pre. In our approach, the formula obtained via $\text{Pre}(I, \psi)$ is immediately simplified to create first ψ', and then $\alpha \Rightarrow \psi'$.

Byte-Addressable Memory. We assumed above that the memory map has type *INT* $\rightarrow$ *INT*. When memory is byte-addressable, the actual memory-map type is *INT32* $\rightarrow$ *INT8*. This complicates matters because accessing (updating) a 32-bit quantity in memory translates into four contiguous 8-bit accesses (updates). For instance, a 32-bit little-endian access can be expressed as follows:

$$\begin{aligned} \textit{access\_32\_8\_LE\_32}(m, a) = \text{let } v4 &= 2^{24} * \textit{Int8To32ZE}(m(a+3)) \\ v3 &= 2^{16} * \textit{Int8To32ZE}(m(a+2)) \\ v2 &= 2^{8} * \textit{Int8To32ZE}(m(a+1)) \\ v1 &= \textit{Int8To32ZE}(m(a)) \\ \text{in } &(v4 \mid v3 \mid v2 \mid v1) \end{aligned} \tag{5.3}$$

[4] A system for DPG needs the symbolic-execution primitive $\text{SE}[\![I]\!]$ anyway for other steps of state-space exploration. Because an implementation of $\text{SE}[\![I]\!]$ can be generated from a description of the semantics of an instruction set ([125] and Chapter 4), an implementation of $\text{Pre}_\alpha(I, \psi)$ can be generated as well.

$$
\begin{array}{l}
\left(\begin{array}{l}
2^{24} * \mathit{Int8To32ZE}(\mathit{ite}(x+3 = p+3, 0, \mathit{ite}(x+3 = p+2, 0, \mathit{ite}(x+3 = p+1, 0, \mathit{ite}(x+3 = p, 5, *(x+3)))))) \\
\mid 2^{16} * \mathit{Int8To32ZE}(\mathit{ite}(x+2 = p+3, 0, \mathit{ite}(x+2 = p+2, 0, \mathit{ite}(x+2 = p+1, 0, \mathit{ite}(x+2 = p, 5, *(x+2)))))) \\
\mid 2^{8} * \mathit{Int8To32ZE}(\mathit{ite}(x+1 = p+3, 0, \mathit{ite}(x+1 = p+2, 0, \mathit{ite}(x+1 = p+1, 0, \mathit{ite}(x+1 = p, 5, *(x+1)))))) \\
\mid \mathit{Int8To32ZE}(\mathit{ite}(x = p+3, 0, \mathit{ite}(x = p+2, 0, \mathit{ite}(x = p+1, 0, \mathit{ite}(x = p, 5, *x)))))
\end{array}\right) \\
+ \left(\begin{array}{l}
2^{24} * \mathit{Int8To32ZE}(\mathit{ite}(y+3 = p+3, 0, \mathit{ite}(y+3 = p+2, 0, \mathit{ite}(y+3 = p+1, 0, \mathit{ite}(y+3 = p, 5, *(y+3)))))) \\
\mid 2^{16} * \mathit{Int8To32ZE}(\mathit{ite}(y+2 = p+3, 0, \mathit{ite}(y+2 = p+2, 0, \mathit{ite}(y+2 = p+1, 0, \mathit{ite}(y+2 = p, 5, *(y+2)))))) \\
\mid 2^{8} * \mathit{Int8To32ZE}(\mathit{ite}(y+1 = p+3, 0, \mathit{ite}(y+1 = p+2, 0, \mathit{ite}(y+1 = p+1, 0, \mathit{ite}(y+1 = p, 5, *(y+1)))))) \\
\mid \mathit{Int8To32ZE}(\mathit{ite}(y = p+3, 0, \mathit{ite}(y = p+2, 0, \mathit{ite}(y = p+1, 0, \mathit{ite}(y = p, 5, *y)))))
\end{array}\right) \\
= 10
\end{array}
$$

Figure 5.2 The formula for Pre(I, ψ), where ψ is *update*$\_32\_8\_LE\_32(M, R(\texttt{ebp}) - 8)$ + *update*$\_32\_8\_LE\_32(M, R(\texttt{ebp}) - 12) = 10$, obtained by evaluating ψ on the symbolic state $S' = [M \mapsto$ *update*$\_32\_8\_LE\_32(M, R(\texttt{eax}), 5), R \mapsto R]$. For brevity, the following notational shorthands are used in the formula: $p = R(\texttt{eax})$, $x = R(\texttt{ebp}) - 8$, $y = R(\texttt{ebp}) - 12$, $*x = M(R(\texttt{ebp}) - 8)$, $*y = M(R(\texttt{ebp}) - 12)$, etc.

where *Int8To32ZE* converts an *INT8* to an *INT32* by padding the high-order bits with zeros, and "|" denotes bitwise-or.

Let *update*$\_32\_8\_LE\_32$ denote the similar operation for updating a map of type *INT32* $\rightarrow$ *INT8* under the little-endian storage convention. Note that when $1 \leq |k_1 -_{\mathit{INT32}} k_2| \leq 3$, we no longer have the property

$$\mathit{access}\_32\_8\_LE\_32(\mathit{update}\_32\_8\_LE\_32(M, k_1, d), k_2) = \mathit{access}\_32\_8\_LE\_32(M, k_2).$$

and hence it is invalid to simplify formulas by the rule

$$\begin{array}{l}\mathit{access}\_32\_8\_LE\_32(\mathit{update}\_32\_8\_LE\_32(M, k_1, d), k_2) \\ \qquad \Rightarrow \mathit{ite}(k_1 = k_2, d, \mathit{access}\_32\_8\_LE\_32(M, k_2)).\end{array}$$

However, the four single-byte accesses on m in Eqn. (5.3) ($m(a)$, $m(a+1)$, $m(a+2)$, and $m(a+3)$) are *access* operations for which it is valid to apply the standard axiom of arrays (i.e., $(m[k_1 \mapsto d])(k_2) = \mathit{ite}(k_1 = k_2, d, m(k_2))$).

Returning to the example discussed above, in which $R(\texttt{eax})$ equals $R(\texttt{ebp}) - 8$ in symbolic state S_n, we perform the same steps as before. First, the symbolic execution of $I =$ `mov [eax],5` starting from the identity symbolic state $S_{id} = [M \mapsto M, R \mapsto R]$ results in the symbolic state

$$S' = [M \mapsto \mathit{update}\_32\_8\_LE\_32(M, R(\texttt{eax}), 5), R \mapsto R].$$

The formula ψ is now written as follows:

$$\mathit{access\_32\_8\_LE\_32}(M, R(\texttt{ebp}) - 8) + \mathit{access\_32\_8\_LE\_32}(M, R(\texttt{ebp}) - 12) = 10.$$

To obtain Pre(I, ψ), we evaluate ψ under S', which yields the formula shown in Fig. 5.2.

The formula shown in Fig. 5.2 is the analog of Eqn. (5.1).

The step that uses symbolic state S_n to identify α and create a simplified formula ψ' that weakens Pre(I, ψ) is now applied to the formula shown in Fig. 5.2 and produces

$$\psi' \stackrel{\text{def}}{=} 5 + \begin{pmatrix} 2^{24} * \mathit{Int8To32ZE}(*(y+3)) \\ \mid\ 2^{16} * \mathit{Int8To32ZE}(*(y+2)) \\ \mid\ 2^{8} * \mathit{Int8To32ZE}(*(y+1)) \\ \mid\ \mathit{Int8To32ZE}(*y) \end{pmatrix} = 10.$$

The α that is the analog of Eqn. (5.2) is the conjunction of the disequalities collected from the formula shown in Fig. 5.2:

$$\begin{aligned} \alpha \stackrel{\text{def}}{=} & \; x+3 \neq p+3 \wedge \ldots x+3 \neq p \wedge \ldots x \neq p+3 \wedge \ldots x \neq p \\ & \wedge\, y+3 \neq p+3 \wedge \ldots y+3 \neq p \wedge \ldots y \neq p+3 \wedge \ldots y \neq p. \end{aligned}$$

As before, the formula $\alpha \Rightarrow \psi'$ is the desired refinement predicate Pre$_\alpha(I, \psi)$.

5.1.2.2 Speculative Trace Refinement

Motivated by the observation that DPG is able to avoid exhaustive loop unrolling if it discovers the right loop invariant, we developed mechanisms to discover candidate invariants from a trace,[5] which are then incorporated into the abstract graph. Although they are only *candidate* invariants, they are introduced into the abstract graph in the hope that they are invariants for the full program. The basic idea is to apply dataflow analysis to a graph obtained from the trace G_π. The recovery of invariants from G_π is similar in spirit to the computation of invariants from traces in Daikon [84], but in MCVETO they are computed *ex post facto* by dataflow analysis on the trace. While any kind of dataflow analysis could be used in this fashion, MCVETO currently uses two analyses:

[5] The trace is *folded* by grouping together all nodes with the same effective address, and augmenting it in a way that overapproximates the portion of the program not explored by the trace (see [174, 175] for more details).

- Affine-relation analysis (§3.3.2 and [141]) is used to obtain linear equalities over registers and a set of memory locations, V. V is computed by running aggregate structure identification [156] on G_π to obtain a set of inferred memory variables M, then selecting $V \subseteq M$ as the most frequently accessed locations in π.
- An analysis based on strided-interval arithmetic (§3.3.4 and [160]) is used to discover range and congruence constraints on the values of individual registers and memory locations.

The candidate invariants are used to create predicates for the nodes of G_π. Because an analysis may not account for the full effects of indirect memory references on the inferred variables, to incorporate a discovered candidate invariant φ for node n into G_π safely, we split n on φ and $\neg\varphi$. Again we have two overapproximations: G_π, from the trace, augmented with the candidate invariants, and the original abstract graph G. To incorporate the candidate invariants into G, we perform $G := G \cap G_\pi$; the $\cap$ operation labels a product state $\langle q_1, q_2 \rangle$ with the conjunction of the predicates on states q_1 of G and q_2 of G_π.

5.1.3 Implementation

The MCVETO implementation incorporates all of the techniques described in §5.1.2. The implementation uses only language-independent techniques; consequently, MCVETO can be easily retargeted to different languages. The main components of MCVETO are language-independent in two different dimensions:

1. The MCVETO DPG driver is structured so that one only needs to provide implementations of primitives for performing concrete and symbolic execution of a language's constructs, plus a handful of other primitives (e.g., Pre_α). Consequently, this component can be used for both source-level languages and machine-code languages.
2. For machine-code languages, we used two tools that *generate* the required implementations of the primitives for concrete and symbolic execution from descriptions of the syntax and concrete operational semantics of an instruction set. The abstract syntax and concrete semantics are specified using TSL. Translation of binary-encoded instructions to abstract syntax

trees is specified using a tool called ISAL (**I**nstruction **S**et **A**rchitecture **L**anguage).[6] The relationship between ISAL and TSL is similar to the relationship between Flex and Bison—i.e., a Flex-generated lexer passes tokens to a Bison-generated parser. In our case, the TSL-defined abstract syntax serves as the formalism for communicating values—namely, instructions' abstract syntax trees—between the two tools.

In addition, we developed language-independent solutions to each of the issues in MCVETO, such as identifying the aliasing condition relevant to a specific property in a given state (§5.1.2.1). Consequently, our implementation acts as a "YACC-like" tool for creating versions of MCVETO for different languages: given a description of language L, a version of MCVETO for L is generated automatically. We created two specific instantiations of MCVETO from descriptions of the Intel x86 and PowerPC instruction sets. To perform symbolic queries on the conceptually-infinite abstract graph (see [174, 175] for details), the implementation uses OpenFst [33] (for transducers) and WALi [114] (for WPDSs).

5.1.4 Experiments

Our experiments (see Fig. 5.15) were run on a single core of a single-processor quad-core 3.0 GHz Xeon computer running Windows XP, configured so that a user process has 4 GB of memory. They were designed to test various aspects of a DPG algorithm and to handle various intricacies that arise in machine code (some of which are not visible in source code). We compiled the programs with Visual Studio 8.0, and ran MCVETO on the resulting object files (without using symbol-table information).[7]

The examples `ex5`, `ex6`, and `ex8` are from the NECLA Static Analysis Benchmarks.[8] The examples `barber`, `berkeley`, `cars`, `efm` are multi-procedure versions of the larger examples on which SYNERGY [98] was tested. (SYNERGY was tested using single-procedure versions only.[9])

[6] ISAL also handles other kinds of concrete syntactic issues, including (a) *encoding* (abstract syntax trees to binary-encoded instructions), (b) *parsing assembly* (assembly code to abstract syntax trees), and (c) *assembly pretty-printing* (abstract syntax trees to assembly code).

[7] The examples are available at `www.cs.wisc.edu/wpis/examples/McVeto`.

[8] `www.nec-labs.com/research/system/systems_SAV-website/benchmarks.php`

[9] `www.cse.iitb.ac.in/~bhargav/synergy`

Program		MCVETO performance (x86)	
Name	Outcome	#Instrs	time
blast2/blast2	timeout	326	**
fib/fib–REACH-0	timeout	287	**
fib/fib–REACH-1	counterex.	287	0.07
slam1/slam1	proof	290	61.85
smc1/smc1–REACH-0*	proof	21	959
smc1/smc1–REACH-1*	counterex.	21	0.016
ex5/ex	counterex.	270	0.18
doubleloopdep/count–COUNT-5	counterex.	252	1.09
doubleloopdep/count–COUNT-6	counterex.	252	1.08
doubleloopdep/count–COUNT-7	counterex.	252	1.21
doubleloopdep/count–COUNT-8	counterex.	252	1.51
doubleloopdep/count–COUNT-9	counterex.	252	2.82
inter.synergy/barber	timeout	454	2.02
inter.synergy/berkeley	counterex.	305	**
inter.synergy/cars	proof	378	5.13
inter.synergy/efm	timeout	403	**
share/share–CASE-0	proof	262	93.95
stress/diamonds–SHORT	proof	257	0.27
cert/underflow	counterex.	323	0.52
instraliasing/instraliasing–REACH-0	proof	46	15.0
instraliasing/instraliasing–REACH-1	counterex.	46	5.86
longjmp/jmp	AE viol.	74	0.015
overview0/overview	proof	49	54.9
small_static_bench/ex5	proof	251	0.13
small_static_bench/ex6	proof	259	1.93
small_static_bench/ex8	proof	297	4.6
verisec-gxine/simp_bad	counterex.	1067	0.094
verisec-gxine/simp_ok	proof	1068	**
clobber_ret_addr/clobber–CASE-4	AE viol.	43	2.13
clobber_ret_addr/clobber–CASE-8	AE viol.	35	0.625
clobber_ret_addr/clobber–CASE-9	proof	35	1.44

Figure 5.3 MCVETO experiments. The columns show whether MCVETO returned a proof, counterexample, or an AE violation (Outcome); the number of instructions (#Instrs); the number of concrete executions (CE); the number of symbolic executions (SE), which also equals the number of calls to the YICES solver; the number of refinements (Ref), which also equals the number of Pre_α computations; and the total time (in seconds). *SMC test case. **Exceeded twenty-minute time limit.

`Instraliasing` illustrates the ability to handle instruction aliasing. (The instruction count for this example was obtained via static disassembly, and hence is only approximate.) `Smc1` illustrates the ability of MCVETO to handle self-modifying code. `Underflow` is taken from a DHS tutorial on security vulnerabilities. It illustrates a `strncpy` vulnerability.

The examples are small, but challenging. They demonstrate MCVETO's ability to reason automatically about low-level details of machine code using a sequence of sound abstractions. The question of whether the cost of soundness is inherent, or whether there is some way that the well-behavedness of (most) code could be exploited to make the analysis scale better is left for future research.

5.1.5 Related Work

Machine-Code Analyzers Targeted at Finding Vulnerabilities. A substantial amount of work exists on techniques to detect security vulnerabilities by analyzing source code for a variety of languages [129, 180, 185]. Less work exists on vulnerability detection for machine code. Kruegel et al. [118] developed a system for automating mimicry attacks; it uses symbolic execution of machine code to discover attacks that can give up and regain execution control by modifying the contents of the data, heap, or stack so that the application is forced to return control to injected attack code at some point after the execution of a system call. Cova et al. [75] used that platform to detect security vulnerabilities in x86 executables via symbolic execution.

Prior work exists on directed *test* generation for machine code [55, 95]. Directed test generation combines concrete execution and symbolic execution to find inputs that increase test coverage. An SMT solver is used to obtain inputs that force previously unexplored branch directions to be taken. In contrast, MCVETO implements directed *proof* generation for machine code. Unlike directed-test-generation tools, MCVETO is goal-directed, and works by trying to refute the claim "no path exists that connects program entry to a given goal state".

Machine-Code Model Checkers. SYNERGY applies to an x86 executable for a "single-procedure C program with only [`int`-valued] variables" [98] (i.e., no pointers). It uses debugging information to obtain information about variables and types, and uses Vulcan [173] to obtain a CFG. It uses integer arithmetic—not bit-vector arithmetic—in its solver. Quoting A. Nori, "[[98] handles] the complexities of binaries via its front-end Vulcan and *not* via its property-checking engine" [150]. In contrast, MCVETO addresses the challenges of checking properties of stripped executables articulated in Chapter 2.

AIR ("Assembly Iterative Refinement") [61] is a model checker for PowerPC. AIR decompiles an assembly program to C, and then checks if the resulting C program satisfies the desired property by applying COPPER [60], a predicate-abstraction-based model checker for C source code. They state that the choice of COPPER is not essential, and that any other C model checker, such as SLAM [47] or BLAST [102] would be satisfactory. However, the C programs that result from their translation step use pointer arithmetic and pointer dereferencing, whereas many C model checkers, including SLAM and BLAST, make unsound assumptions about pointer arithmetic.

[MC]SQUARE [165] is a model checker for microcontroller assembly code. It uses explicit-state model-checking techniques (combined with a degree of abstraction) to check CTL properties.

Our group developed two prior machine-code model checkers, CodeSurfer/x86 [44] and DDA/x86 [43]. Neither system uses either underapproximation or symbolic execution. For over-approximation, both use numeric static analysis and a different form of abstraction refinement than the one used in MCVETO.

Self-Modifying Code. The work on MCVETO addresses a problem that has been almost entirely ignored by the PL research community. There is a paper on SMC by Gerth [90], and a recent paper by Cai et al. [59]. However, both of the papers concern proof systems for reasoning about SMC. In contrast, MCVETO can verify (or detect flaws in) SMC automatically.

As far as we know, MCVETO is the first model checker to address verifying (or detecting flaws in) SMC.

5.1.6 Conclusion

MCVETO resolves many issues that have been unsoundly ignored in previous work on software model checking. MCVETO addresses the challenge of establishing properties of the machine code that actually executes, and thus provides one approach to checking the effects of compilation and optimization on correctness. The contributions of the work described in §5.1.2 lie in the insights that went into defining the innovations in dynamic and symbolic analysis used in MCVETO: (i) sound disassembly and sound construction of an overapproximation (even in the presence of instruction aliasing and self-modifying code) (see [174] for the details), (ii) a new method to

eliminate families of infeasible traces (see [174] for the details), (iii) a method to speculatively, but soundly, elaborate the abstraction in use (§5.1.2.2), (iv) new symbolic methods to query the (conceptually infinite) abstract graph (see [174] for the details), and (v) a language-independent approach to Pre_α (§5.1.2.1). Not only are our techniques language-independent, the implementation is parameterized by specifications of an instruction set's semantics. By this means, MCVETO has been instantiated for both x86 and PowerPC.

5.2 BCE

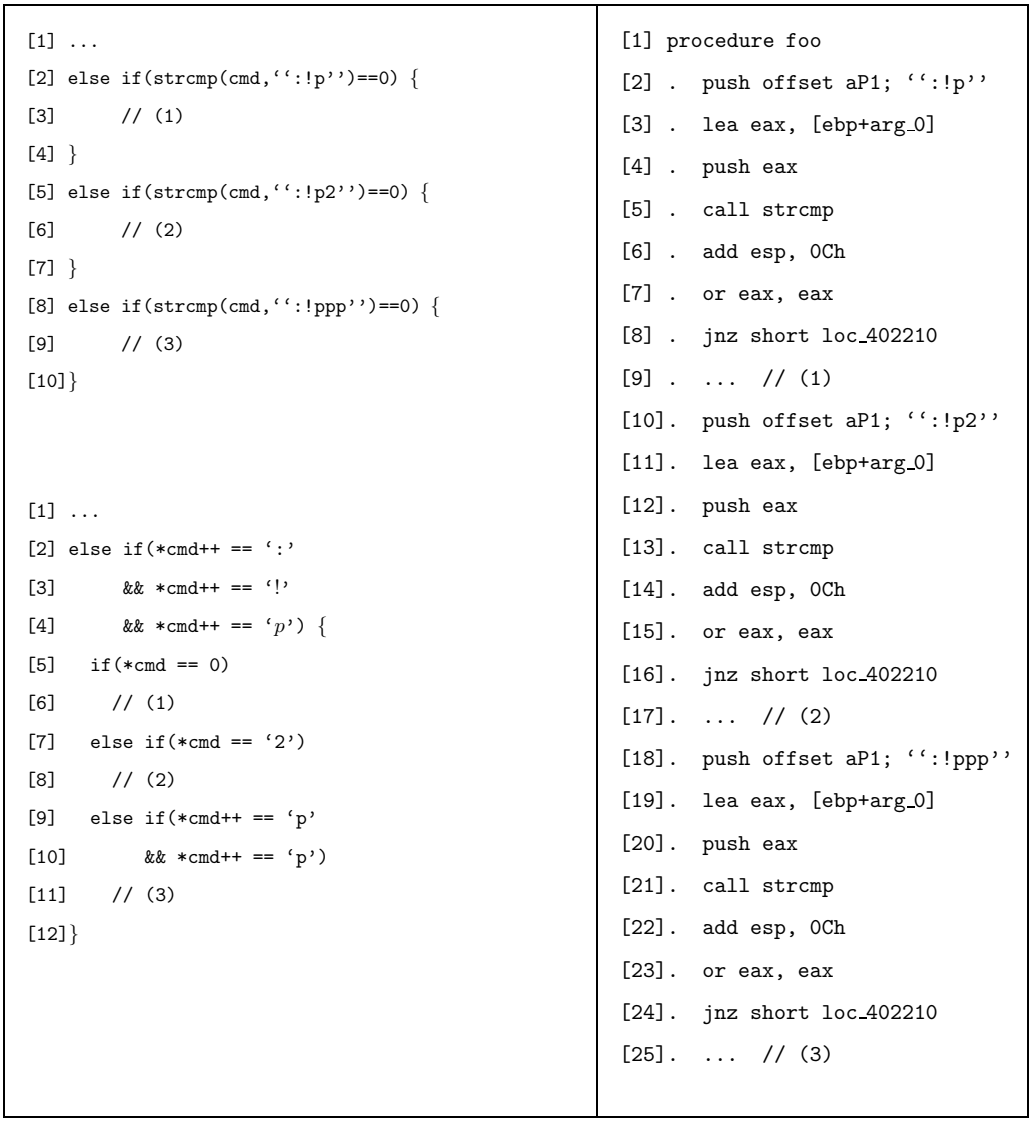
As discussed in §1.5.4, an increasing number of individual Internet sites have been compromised by attacks from across the world to become part of various kinds of malicious botnets. The Internet security research community has made significant efforts to identify botnets, to collect data on their activities, and to develop techniques for detection, mitigation, and disruption.

We have developed a tool called BCE (Botnet-Command Extractor) for extracting botnet-command information from bot executables. BCE aims to provide useful information from analysis of bot executables by automatically extracting proper inputs that trigger malicious behavior. Applications of the information recovered include observing and analyzing malicious behaviors, as well as identifying and mitigating botnets.

A typical way to analyze the behavior of a bot is to run the executable and observe its actions. To carry this out, however, one needs proper inputs to trigger malicious behaviors. Some widely-known commands are often used for this purpose. However, attackers can easily change their commands to evade such dynamic analysis. Also, it is a hard problem to obtain such inputs by manually stepping through the executable. BCE automates the extraction of information about botnet commands and the arguments to commands.

The work described in the section makes the following contributions:

1. BCE automatically extracts botnet-command information from bot executables, without source code or symbol-table/debugging information. The extracted information includes (a) constant command strings that trigger API-level behaviors, (b) relationships, including type relationships, between the input command string and the actual parameters of an API



```
[1] ...
[2] else if(strcmp(cmd,‘‘:!p’’)==0) {
[3]      // (1)
[4] }
[5] else if(strcmp(cmd,‘‘:!p2’’)==0) {
[6]      // (2)
[7] }
[8] else if(strcmp(cmd,‘‘:!ppp’’)==0) {
[9]      // (3)
[10]}
```

```
[1] ...
[2] else if(*cmd++ == ‘:’
[3]      && *cmd++ == ‘!’
[4]      && *cmd++ == ‘p’) {
[5]   if(*cmd == 0)
[6]     // (1)
[7]   else if(*cmd == ‘2’)
[8]     // (2)
[9]   else if(*cmd++ == ‘p’
[10]       && *cmd++ == ‘p’)
[11]    // (3)
[12]}
```

```
[1] procedure foo
[2] .  push offset aP1; ‘‘:!p’’
[3] .  lea eax, [ebp+arg_0]
[4] .  push eax
[5] .  call strcmp
[6] .  add esp, 0Ch
[7] .  or eax, eax
[8] .  jnz short loc_402210
[9] .  ...  // (1)
[10].  push offset aP1; ‘‘:!p2’’
[11].  lea eax, [ebp+arg_0]
[12].  push eax
[13].  call strcmp
[14].  add esp, 0Ch
[15].  or eax, eax
[16].  jnz short loc_402210
[17].  ...  // (2)
[18].  push offset aP1; ‘‘:!ppp’’
[19].  lea eax, [ebp+arg_0]
[20].  push eax
[21].  call strcmp
[22].  add esp, 0Ch
[23].  or eax, eax
[24].  jnz short loc_402210
[25].  ...  // (3)
```

Figure 5.4 (a) (top left) A snippet of the EvilBot source code, (b) (bottom left) alternative source code, (c) (right) the assembly code of (a).

call, and (c) constraints on the actual parameters of an API call. The information obtained via BCE can be used to build up proper input commands that trigger API-level behaviors.

2. BCE is able to provide a specification of the API-level behaviors of a bot program without running the bot. Along with the input-command strings extracted from a bot program, BCE also provides a sequence of API calls controlled by each command, which can help the user understand the API-level behavior.

3. BCE is not based on signatures. Some recent approaches to finding out botnet commands are based on pattern-matching techniques. Many bot programs use standard string-library functions to process the input command string, as shown in Fig. 5.4(a). The assembly code of Fig. 5.4(a) obtained using the IDAPro disassembler is shown in Fig. 5.4(c). One can find a pattern in the assembly code: there are two `push` instructions, one of which is for a constant string that IDApro readily identifies, followed by a call to `strcmp`. However, such a technique is ad hoc and can be easily evaded, e.g., by changing the code in Fig. 5.4(a) to use byte-by-byte comparison instead of using standard library functions, as shown in Fig. 5.4(b).
4. BCE uses directed test generation [94], enhanced with a new search technique that uses control-dependence information [86] to direct the search. Our experiments show that the method provides higher coverage of the parts of the program relevant to identifying bot commands, as well as lower overall execution time than the standard program exploration that does not use control-dependence information.
5. We performed experiments with four real bot programs. Our preliminary results show that BCE is able to effectively extract bot-command information.

Organization. The remainder of the section is organized as follows: §5.2.1 discusses what kind of information BCE extracts, and how one can make use of the information to trigger potentially malicious behaviors from a bot. §5.2.2 presents background on directed test generation [94]. §5.2.3 presents the enhanced techniques for exploring program paths that we developed for use in BCE. §5.2.4 describes the use of nondeterminism in BCE, which is used for writing "harness" code to model possible client environments, possible inputs, and possible return values from library functions or system calls. §5.2.5 discusses additional information that BCE recovers, which combines the recovered information about constraints on inputs with type information for the target API calls. §5.2.6 describes how a language-independent BCE implementation was created. §5.2.7 presents experimental results. §5.2.8 discusses the limitations of BCE. §5.2.9 discusses related work. §5.2.10 concludes.

5.2.1 Botnet-Command Extractor (BCE)

In this section, we first discuss what information BCE relies on to extract botnet commands. We then summarize the kind of information that BCE provides, and how one can make use of such information to generate proper input commands.

5.2.1.1 What BCE Relies On

1. API prototypes: BCE relies on information about function prototypes of API functions (system calls). For example, the prototype of *ShellExecute* is as follows:

```
HINSTANCE ShellExecute(
HWND hwnd,
LPCTSTR lpOperation,
LPCTSTR lpFile,
LPCTSTR lpParameters,
LPCTSTR lpDirectory,
INT nShowCmd
);
```

`lpDirectory`: *[in] A pointer to a null-terminated string that specifies the default (working) directory for the action.*

   The function prototypes are used to construct reasonable input commands given the command specification extracted by BCE.

2. Control-Dependence Graph: BCE makes use of the control-dependence graph for a bot binary to optimize its state-space-exploration algorithm. We discuss the use of control dependences in more detail in §5.2.3.

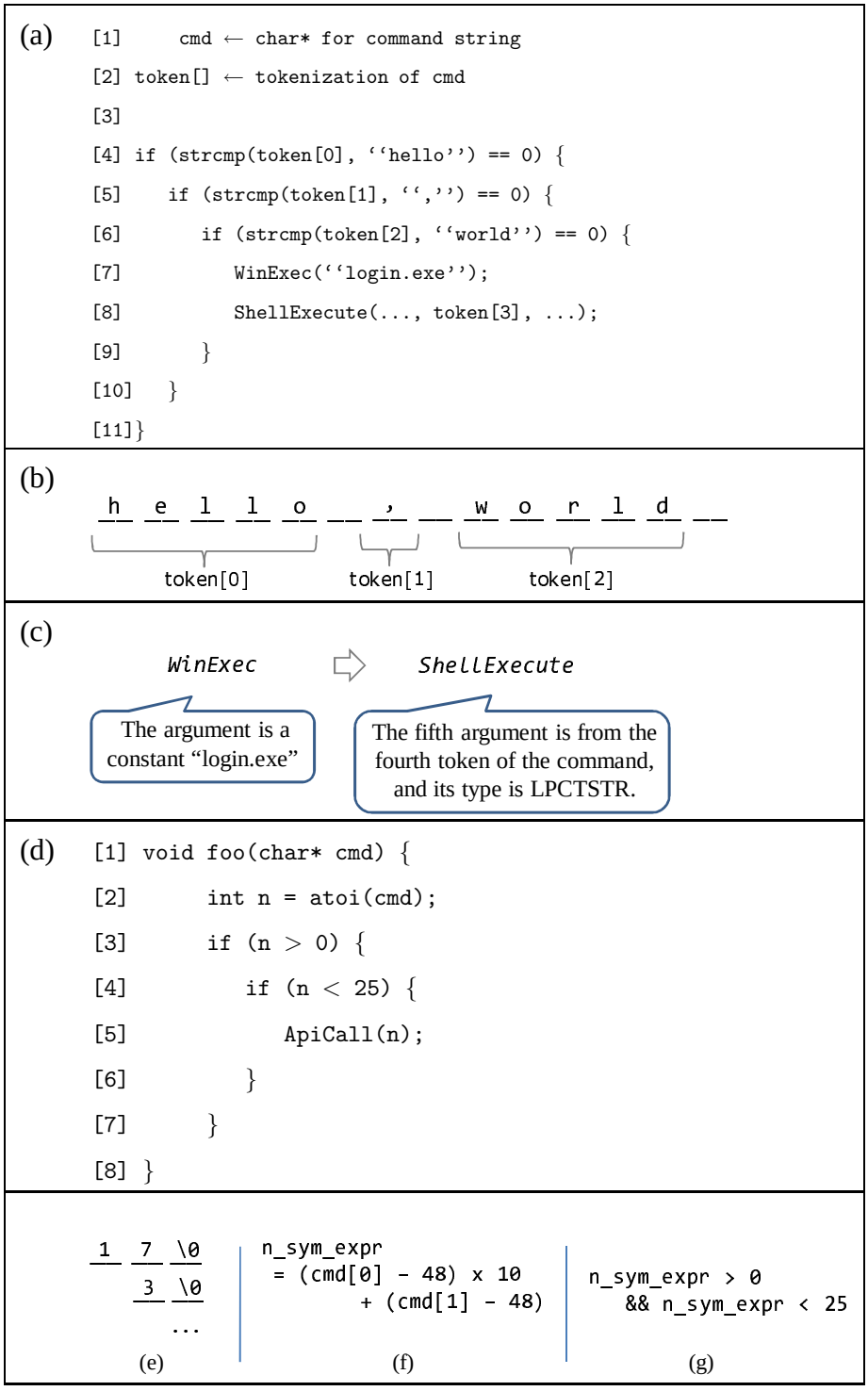


Figure 5.5 (a) A simple example program; (b) the command string constructed based on the information obtained from BCE; (c) a sequence of API calls obtained from BCE; (d) another simple example; (e) constant examples provided by BCE; (f) the symbolic expression obtained from BCE for the argument n; (g) the constraint obtained from BCE.

5.2.1.2 What BCE Recovers and How to Use the Recovered Information

1. Constant command strings that control a bot. For example, there are three nested if-statements in the code shown in Fig. 5.5(a). Two API calls are invoked when the three branch conditions are satisfied. Suppose that `cmd` has been tokenized into three null-terminated strings. Fig. 5.5(b) is the command string constructed based on the information extracted by BCE. This information is obtained from conditional branches where a portion of the command string is compared against some constants, as the three strings ("hello", ",", and "world") in the example.

2. A sequence of API calls controlled by each command. Along with each command, BCE provides a sequence of API calls that are controlled by the command. For example, the code executed when the command string shown in Fig. 5.5(b) is issued subsequently invokes *WinExec* and *ShellExecute*. This information can be directly used to get an idea of the API-level behavior of a bot without actually executing it.

3. Information about the actual arguments of each API call. In addition to a sequence of API calls, BCE provides information about the arguments to each API call, such as constant values for an argument, symbolic expressions, and constraints on the symbolic expressions, as shown in Fig. 5.5(e), (f), and (g), respectively.

- *Constant arguments:* In many cases, API calls take constant arguments that one can statically extract from binaries. For example, the first argument of *WinExec* in Fig. 5.5(a) is a constant string "login.exe". In addition to the sequence of API calls, information about argument values enables one to get a better idea of the API-level behavior of a bot without running it.
- *Symbolic expressions in the input-state vocabulary:* BCE also provides a symbolic expression for each actual parameter of an API call, along with its type information, as long as the argument is related to some part of the input command. For example, *ShellExecute* in Fig. 5.5(a) takes the fourth token of the input command as its fifth argument. BCE automatically extracts a symbolic expression that has one symbolic

term, `token[3]`, along with its type LPCTSTR. The type information is obtained from the prototype of the API call. The type information is used to come up with a proper input string. Given the information that the fourth token is supposed to be a null-terminated string that specifies a working directory name, one can build up a complete command string as follows:

```
"hello , world C:\temp"
```

Fig. 5.5(f) shows another example of a symbolic expression that BCE provides. Fig. 5.5(f) is the symbolic expression obtained for n in Fig. 5.5(d). In Fig. 5.5(d), the input command string is a numeral, which is converted into a number by calling `atoi`; the number is then passed into an API call as an argument. The symbolic expression is in the *input vocabulary* in that the symbols (`cmd[0]` and `cmd[1]`) that appear in it represent individual byte values of the input command string. We discuss how the symbolic expression is generated in §5.2.2.

- *Constraints on symbolic expressions:* BCE also provides constraints on the symbolic expressions extracted for each actual parameter of an API call, if any. For example, BCE extracts the constraint shown in Fig. 5.5(g) for the actual parameter n to the API call in Fig. 5.5(d).

  This constraint is obtained from the two conditional branches that guard the API call. BCE finds out the conditional branches on which the API call transitively depends. It only collects branches whose predicates constrain the given symbolic expression.

  The obtained constraints also play an important role for building up proper input commands. BCE provides some concrete examples for n, as shown in Fig. 5.5(e): the numeral strings "17" and "3" satisfy the two branch predicates ($n > 0$ and $n < 25$). Therefore, these input strings cause the API call to be invoked, and thus can be directly used to run the bot program. However, there are cases when the automatically generated concrete examples fail to trigger observable behavior of a bot. For example, suppose the API in Fig. 5.5(d) is some API that takes an IP address and sets up a connection to the server (e.g., `httpserver` of SpyBot). Because concrete examples are randomly

selected to satisfy the constraints collected during symbolic execution, it is not likely that BCE finds out a reasonable IP address unless there are conditional branches where it can extract proper constraints on the command. Therefore, in some cases, the user is responsible for making use of the extracted constraints to construct reasonable inputs.

§5.2.5 discusses other kinds of information about the bot's commands that BCE provides—in particular, information that combines the recovered symbolic information about inputs with type information for the target API calls.

5.2.2 Background on Directed Test Generation and Overview of BCE

This section provides background on *directed test generation* [94], which collects path constraints and uses them to explore new paths systematically. In applying directed test generation in BCE to the problem of extracting bot commands, we developed new techniques to explore program paths, which differ from conventional directed-test-generation techniques. We discuss our enhanced search algorithms in §5.2.3.

One example of a directed test-generation tool is SAGE [95], which is a whitebox fuzz-testing tool, an advance on fuzz testing based on random mutations. SAGE records an actual run of a program under test, starting with a well-formed input, then symbolically evaluates the recorded trace and generates constraints that capture how the program uses its inputs. The generated constraints are then systematically modified and solved with a constraint solver to produce new inputs that cause the program to follow different control-flow paths. The process is repeated with a coverage-maximizing heuristic designed to find defects as fast as possible. Fig. 5.6 shows a simple example taken from [95]. There are 5 values leading to the error out of 2^{8*4} possible values for 4 bytes. Therefore, the probability of hitting the error with random testing is about $1/2^{32}$. In contrast, whitebox dynamic test generation can find the error in at most $2^4 = 16$ iterations (4 valid path constraints are collected during the exploration process).

Algorithm 2 Single BCE Iteration

Require: A concrete state S.

Require: A trace tree T

1: Concretely execute the program with the concrete state S.
2: Let CT be the concrete trace obtained from the concrete execution.
3: Symbolically execute the trace CT.
4: Let T' be the trace tree augmented by the symbolic execution.
5: **if** at least one API call is encountered in the concrete trace **then**
6: Based on the symbolic state obtained in the symbolic execution, collect information about the command tokens that appear in the arguments to each API call.
7: **end if**
8: **repeat**
9: Choose a new path π in the trace tree T.
10: Let φ be the path-constraint formula obtained by conjoining the branch constraints along π.
11: **until** φ is satisfiable
12: Let M be the model obtained by calling the constraint solver with φ.
13: Create the new concrete state S' updated with the assignments from the model M.

Alg. 2 shows the basic search step of the BCE algorithm. The outline of the algorithm is similar to typical directed-test-generation techniques, which can be roughly summarized as repeatedly applying the following three steps:[10]

BCE maintains a trace tree that is expanded during the process of symbolic execution. Each node in a trace tree represents a different execution instance of a branch instruction in the program. Each node can have two children, one of which represents the first branch node encountered along the path through the true successor, the other of which is the first branch node along the path through the false successor. The path from the root node to a leaf node represents the branch

[10]The first step (concrete execution) and the second step (symbolic execution) can be done simultaneously, which is sometimes called *concolic execution* [167]. In concolic execution, concrete values from the concrete execution state are sometimes used to simplify the symbolic states created during symbolic execution.

```
void top(char input[4]) {
  int cnt = 0;
  if (input[0] == 'b') cnt++;
  if (input[1] == 'a') cnt++;
  if (input[2] == 'd') cnt++;
  if (input[3] == '!') cnt++;
  if (cnt >= 4) abort();
}
```

Figure 5.6 An example for whitebox fuzz testing

instructions of a concrete trace. Each edge holds a branch constraint obtained from symbolic execution. Each time a branch is symbolically executed (to follow the direction taken by a previous concrete execution), the trace tree is extended appropriately.

5.2.3 Program Exploration using Control-Dependence Information

This section presents the enhanced techniques for exploring program paths that we developed for use in BCE. MineSweeper [55] and the work of Moser et al. [139] have shown the potential for carrying out better exploration in malware. Other tools, such as SAGE, have addressed the problem of path explosion by introducing heuristics to improve coverage [95]. SAGE uses so-called *generational search* designed to partially explore the state spaces of large applications with the aim of finding bugs faster. As in most of other directed-test-generation tools, SAGE aims to improve test coverage. Unlike bug-finding tools or tools that aim to improve coverage, in BCE we are interested in *goal-directed techniques* aimed at extracting bot commands.

The characteristics of how the bot code parses the transmitted commands and takes actions depending on the parsed commands can be used to come up with better exploration strategies that avoid possible explosion and obtain more complete specifications about the command structure. We incorporated the following path-exploration strategies into BCE:

- Choose as a candidate for the new path the branches that have a possibility of leading to API calls.[11]
- Prune the search performed by BCE so that each path includes a limited number of API calls if a candidate branch for extending the path is independent of the branches involved with the API calls already found in the path.

The exploration strategies are based on the fact that our goal is to identify as many feasible input commands as possible that lead to API calls of interest.

To identify branches that have a possibility of encountering API calls, we use *control-dependence information*. §5.2.3.1 discusses control-dependence information. In §5.2.3.2 and §5.2.3.3, we present how control-dependence information is used in BCE.

5.2.3.1 Control Dependence

The *control dependence* relation is one of the fundamental relationships among statements or instructions used in compilers and optimizers. For instance, control-dependence information is used in compilers to determine whether it is safe to reorder or parallelize statements [86]. A control dependence holds when the decision made at a branch X controls whether another statement or instruction Y is executed.

Control dependence is defined in terms of the post-domination relation.

Definition 5.1 Node Z *post-dominates* node X iff $Z \neq X$ and all paths from X to the end of the procedure include Z. (Note that by this definition a node does not post-dominate itself.)

Definition 5.2 Node Y is *directly control dependent on node* X iff

1. there exists a path π: $X \rightarrow^{+} Y$ such that Y post-dominates every node in π different from X, and
2. X is not post-dominated by Y.

We use C to denote the direct-control-dependence relation.

[11] BCE is parameterized to take a list of interesting API entry points of interest.

Control dependences can be broken down more finely into dependences on the true branch or false branch of a branch-node X, as follows:

Definition 5.3 Node Y is *directly control-dependent on edge* $X \rightarrow W$ iff

1. there exists a path π: $W \rightarrow^* Y$ such that Y post-dominates every node in π different from X, and
2. X is not post-dominated by Y.

We say that the relation $C_t(X, Y)$ holds when X is a branch node and Y is directly control dependent on X's true branch. C_f is defined similarly.

Each branch node is associated with two sets of CFG nodes: one consists of the transitive control-dependence successors for its true branch (denoted by C_tC*); the other consists of the transitive control-dependence successors for its false branch (denoted by C_fC*).

$$C_tC* : \text{True control successors}$$
$$C_fC* : \text{False control successors}$$

For example, in Fig. 5.7, the statements `(s1)` and `(s2)` are transitively control dependent on the true branch of `b1`; statement `(s3)` is transitively control dependent on the false branch of `b1`. Statement `(s4)` is not transitively control dependent on any branch in this example. (Henceforth, we will abbreviate "transitive control dependence" by "control dependence".)

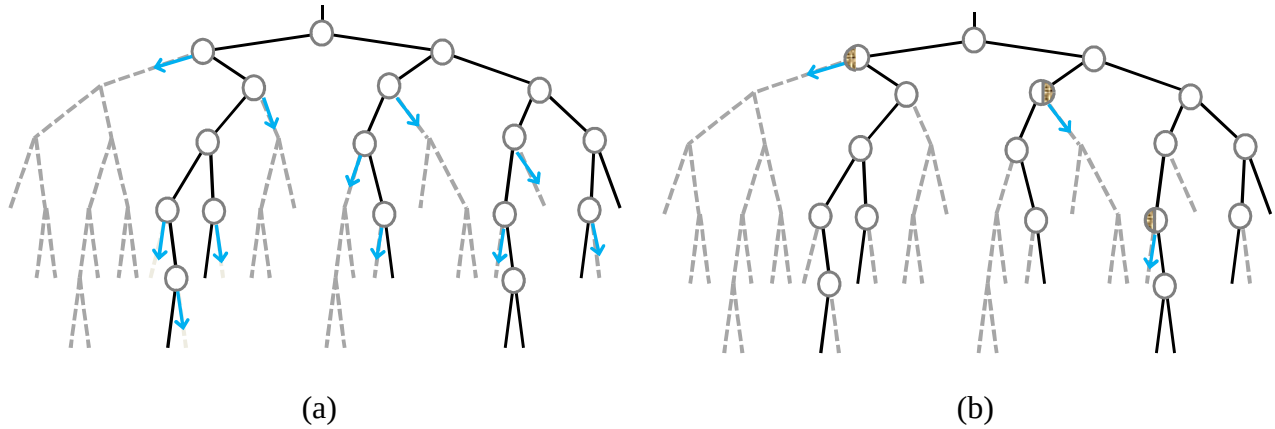
In the next section, we discuss a novel usage of control-dependence information in BCE.

5.2.3.2 Choosing Interesting Branches using Control-Dependence Information

BCE uses control-dependence information (CDI) to annotate the trace tree. If there is at least one API call in C_tC* (or C_fC*) of a branch node, the node is marked as N_t (or N_f). Any branch that has a call to a function that contains at least one N_t or N_f in C_tC* (C_fC*) is also marked as N_t (or N_f). BCE only chooses one of the nodes marked with N_t or N_f as a candidate for the new path. Fig. 5.8 compares an exploration strategy that uses control-dependence information (CDI) to one that does not. The solid lines in the figures indicate the paths that have previously been

```
[1] if (a > 0) { // (b1)
[2]    b = 1; // (s1)
[3]    if (a < 25) { // (b2)
[4]       c = 2; // (s2)
[5]    }
[6] }
[7] else {
[8]    d = 3; // (s3)
[9] }
[10]e = 4; // (s4)
```

Figure 5.7 An example to show control dependences.



(a) (b)

Figure 5.8 Two trace trees; (a) A trace tree without CDI; (b) a trace tree with CDI; the circles represent branch nodes; the solid arrows represent possible paths to explore; the half-shaded circles represent nodes labeled as either N_f or N_t.

explored. One chooses as the next candidate one of the nodes (on the solid lines in Fig. 5.8) that has a solid edge to only one child. Such choices are marked with solid grey arrows. There are fewer candidates to explore in Fig. 5.8(b) than in Fig. 5.8(a). The degree of the improvement by using CDI depends on the percentage of nodes marked with N_t or N_f. We discuss how the approach works out with real bot programs in §5.2.7.

```
[1]  char* p1; // input;
[2]  char p2[] = "bot.execute";
[3]  int v;
[4]  char c1;
[5]  do {
[6]    c1 = *p1++;
[7]    c2 = *p2++;
[8]    v = (unsigned)c1 - (unsigned)c2;
[9]    if(v != 0)
[10]      break;
[11] } while(c1 != '\0');
[12]
[13] if(v == 0)
[14]    APICall
```

Figure 5.9 An example in which it is necessary to choose an alternative candidate as a new path; the source code of `strcmp` is inlined in this example.

Algorithm 3 ChooseNewPath

Require: A trace tree T

Ensure: Formula φ

1: Let Frontier be the branch node in T that is either marked as N_f and does not have a false child in T, or marked as N_t and does not have a true child in T, and has the shortest path from the root node.

2: Let φ be the formula conjoined with all the formulas associated with the branches on the path from Frontier back to the root node.

3: Return φ

Algorithms. Alg. 3 and Alg. 4 describe the path-exploration algorithm of BCE. In Alg. 3, BCE chooses a node n in the trace tree marked as N_f or N_t whose corresponding branch is not in the trace tree. BCE then conjoins all the formulas of the branches on the path from n back to the root node. Alg. 4 takes that formula and calls a constraint solver to obtain a model. If the formula for

Algorithm 4 GenerateNewConcreteState

Require: A trace tree T

Ensure: A concrete state CS$'$

```
1: φ = ChooseNewPath(T)
2: Call the constraint solver with the formula φ
3: if φ is feasible then
4:     Let M be the model from the constraint solver
5:     Let CS be a random concrete state
6:     Let CS' be CS updated with all the assignments in M
7:     Return CS'
8: else
9:     Let T' be T augmented with a dummy node at the previously selected node
10:    GenerateNewConcreteState(T')
11: end if
```

the path that BCE chose to explore is feasible, it generates a new concrete state that gets used in the next round of exploration. Otherwise, it augments the trace tree so that the previously explored path is never selected again, and calls itself recursively.

Fig. 5.10(a) is an example in which the number of possible execution paths is exponential in the number of branches: each of the 5 `if`-statements is independent of each other. For this code fragment, BCE takes 8 iterations when it uses CDI,[12] of Alg. 2 to identify 2 different paths (one toward the API call inside the second `if`-statement, and the other toward the fifth statement) whereas without CDI it exhibits exponential behavior.

Indirect control-dependence. In some cases, it is possible that a candidate node marked as N_t or N_f has a branch predicate, the negation of which causes the path constraint to be infeasible, that does not help program exploration. For example, in Fig. 5.9, `p1` points to the input character array, and `p2` points to the constant string `"bot.execute"`. The branch on line 13 is marked as N_t

[12]The body of `strcmp` includes some branches to compare an individual character of the first argument with one constant character from the second argument. To get to the two API call sites, BCE needs several trials for each.

```
[1]  if(strcmp(c[0], "aaa")==0) {
[2]    n = atoi(c[5]);
[3]  }
[4]  if(strcmp(c[1], "bbb")==0) {
[5]    APICall1(...);
[6]  }
[7]  if(strcmp(c[2], "ccc")==0) {
[8]    n = atoi(c[5]);
[9]  }
[10] if(strcmp(c[3], "ddd")==0) {
[11]   n = atoi(c[5]);
[12] }
[13] if(strcmp(c[4], "eee")==0) {
[14]   APICall2(...);
[15] }
```

(a)

```
[1]  if(strcmp(c[0], "aaa")==0) {
[2]    n = atoi(c[5]);
[3]  }
[4]  else if(strcmp(c[1], "bbb")==0) {
[5]    APICall1(...);
[6]  }
[7]  else if(strcmp(c[2], "ccc")==0) {
[8]    n = atoi(c[5]);
[9]  }
[10] else if(strcmp(c[3], "ddd")==0) {
[11]   n = atoi(c[5]);
[12] }
[13] else if(strcmp(c[4], "eee")==0) {
[14]   APICall2(...);
[15] }
```

(b)

Figure 5.10 (a) An example with independent `if`-statements (and thus an exponential number of paths). (b) An example more typical of bot code (with a linear number of paths).

because its true branch contains an API call. Suppose that in the initial concrete state, the first input byte pointed to by `p1` is something different from `'b'`, and thus the loop in lines 5–11 terminates at line 9 after one iteration with the condition `v != 0`, and the false branch of line 13 is executed. In the subsequent symbolic execution in which the character array pointed to by `p1` is treated as a list of symbols, the path constraint toward the true branch at line 13 is

$$(S_{c1} - C_b \neq 0) \ \wedge \ (S_{c1} - C_b = 0),$$

where S_{c1} is a symbol that represents the first input byte, and C_b is a constant symbol. This formula is infeasible. In such cases, as a heuristic, BCE chooses branches prior to the candidate node on the trace as an alternative candidate. In this example, the false branch at line 9 is chosen as a new path so that from the path constraint

$$S_{c1} - C_b = 0,$$

the constraint solver can provide a new test input in which the first input byte equals $'b'$.

When a situation occurs like the one described for line 13, a command-line flag controls how many prior branches to try.

5.2.3.3 Pruning the Trace Tree using Control-Dependence Information

CDI helps to direct program exploration toward API call sites. However, even when some candidate branches are excluded by CDI, there is still the possibility of combinatorial explosion. For example, in Fig. 5.10(a), there are 24 paths in total that invoke the API call(s): there are 8 paths that invoke each call (and not the other) and an additional 8 that invoke both. When the branches controlled by different commands are independent of each other, it means that multiple commands can be combined to produce different sequences of API calls. In other words, if there are n independent `if`-statements involved with API calls, the total number of possible paths that invoke at least one API call is 2^n.

To avoid such combinatorial explosion, we limit the exploration performed by BCE so that each path includes a limited number of API calls if a candidate branch for extending the path is independent of the branches involved with the API calls already found in the path. In particular, the path exploration in BCE only finds n paths when there are n independent `if`-statements involved with API calls. The information obtained in this way is still useful to a user, although it shifts the burden onto the user to identify the API-level behaviors of a bot by trying various combinations of the n extracted commands. For the example in Fig. 5.10(a), BCE only extracts

"bbb" for the second token of cmd
"eee" for the fifth token of cmd

and the user can try running the bot with the three kinds of inputs—"bbb", "eee", and "bbb" + "eee"—to observe possibly different behaviors.

The heuristic for avoiding combinatorial explosion is performed by pruning the trace tree dynamically. The following code illustrates what is involved in dynamically pruning the trace tree.

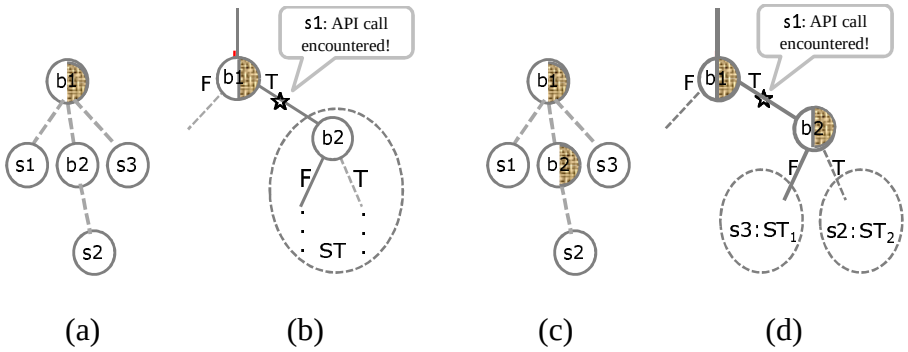


Figure 5.11 (a) A control-dependence graph; (b) a trace tree when sub-trees are pruned using control-dependence graph (a); (c) another control-dependence graph; (d) the trace tree when sub-trees are pruned using control-dependence graph (c).

Fig. 5.11(a) is the control-dependence graph of the code, and Fig. 5.11(b) is the corresponding trace tree.

```
[1] if (strcmp(token[0], ‘‘hello’’) == 0) {
[2]    APICall1(...)
[3]    if (atoi(token[1]) > 0)
[4]       ...
[5]    ...
[6] }
```

Figure 5.12 A simple example for pruning.

An API call is invoked immediately in the true branch of line 1 in Fig. 5.12. In this case, BCE considers pruning the sub-tree ST of the trace tree starting from line 3. The control-dependence information is used to determine whether the sub-tree ST is to be excluded from further exploration. ST can be excluded if it does not include any node marked as N_t or N_f that is control dependent on line 3 (see Fig. 5.11(b)). If there is at least one other API call in line 4, as shown in Fig. 5.11(c) and (d), the true branch remains as a candidate to explore because the second `if`-statement is control dependent on the first one.

In practice, many bot programs are written as shown in Fig. 5.10(b), where each `if`-statement is dependent on other ones. However, even if when they are rewritten in the form of Fig. 5.10(a), the pruning technique is effective in practice.

5.2.4 Using Nondeterminism to Sidestep System Calls

Many formalisms for symbolic analysis of programs support the use of nondeterminism, which is useful for writing "harness code" (code that models the possible client environments from which the code being analyzed might be called), as well as for modeling the possible inputs to a program. A common approach is to provide a primitive that returns an arbitrary value of a given type. Examples include the `SdvMakeChoice` primitive of SLAM [46] and the `havoc(x)` primitive of BoogiePL [48].

In some cases, a value returned from a system call or a Windows-API call is used in a branch condition, as shown in Fig. 5.13. If *GetCurrentDirectory* returns a value greater than 0, `APICall1` is invoked; otherwise, `APICall2` is invoked.

```
[1] for (i = 0; i < 3; i++) {
[2]     int n = GetCurrentDirectory(...);
[3]     if (n > 0) {
[4]         APICall1(...)
[5]     }
[6]     else {
[7]         APICall2(...)
[8]     }
[9] }
```

Figure 5.13 A simple example for modeling a system call.

In the current version of BCE, concrete execution and symbolic execution do not go into system calls and Windows API functions. Instead, BCE keeps a sequence of random numbers (*RandSeq*) for concrete execution, and a sequence of symbols ($\overline{\textit{RandSeq}}$) for symbolic execution. During

concrete execution and symbolic execution, the successive values in *RandSeq* and $\overline{RandSeq}$, respectively, are used as the successive return values from API call sites. In the above example, there are three calls to *GetCurrentDirectory* in a trace because the loop is executed three times. Each of the three return values comes from successive elements of *RandSeq* and $\overline{RandSeq}$. In this way, we model the state of the operating system. Network inputs are modeled similarly.

5.2.5 Extracting Type Information

§5.2.1 briefly discussed how one can use the information extracted from BCE to understand a bot program and construct proper input commands. This section discusses some additional information that BCE provides to help users understand the recovered information about the botnet's commands, based on combining the recovered symbolic information about inputs with type information for the target API calls.

Some extracted constant command strings can be directly used to trigger interesting API-level behaviors of a bot program in cases where there are no additional arguments to a command. However, some of the information extracted about a command is in the form of *symbolic expressions*. A symbolic expression captures the semantics of all the instructions on a specific path from the starting point to the API call site. In some cases, the extracted symbolic expression simply represents a sub-string of the command, whereas there are other cases when the command is converted to another form. A typical action is to convert part of the input string, using the standard library function *atoi*, into a number that is passed to the API call. In other words, the input string holds numerals, whereas the API call receives a number.

Once BCE extracts a symbolic expression for an argument to an API call, it is the user's responsibility to choose a proper input with which to run the bot based on the symbolic expression. To help in this step, BCE extracts type information for each symbolic expression using the algorithms shown in Alg. 5 and Alg. 6.

Alg. 5 and Alg. 6 are pseudo-code for collecting type information for each extracted symbolic expression. Our approach uses information about the function prototypes of API calls, as well as a database of OS and network-related types. For example, Fig. 5.14(a) shows the prototype of

Algorithm 5 ExtractTypeInformation

Require: A function prototype T

Require: A symbolic state S

Require: The current stack address *sp*

Ensure: Updated database

1: Let N be the number of arguments of function type T
2: **for** $i = 0$ to $N - 1$ **do**
3: Let T_i be the type of the i^{th} argument of function type T
4: $addr_i = sp + i * param\_size$
5: CollectTypeInformation(T_i, $addr_i$)
6: **end for**

`getaddrinfo` and the `struct` types `ADDRINFO` and `sockaddr_in`. `ADDRINFO` is the type of the third and fourth arguments of `getaddrinfo`, and `sockaddr_in` is the type of one of the fields of `ADDRINFO`.

For each API call site, BCE collects type information by calling *ExtractTypeInformation* (Alg. 5). Along with such information, *ExtractTypeInformation* takes the symbolic state at the API call site, and the symbolic expression that represents the current stack pointer. For example, Fig. 5.14(b) is an example that includes a call to the system call `getaddrinfo`. The first token of the command is converted to a numeric value through `atoi` to be used as `sin_zero` for the `sockaddr_in` object, and the second token is used as `ai_canonname` for the `ADDRINFO` object. BCE calls *CollectTypeInformation* with the actual arguments—`ADDRINFO*` and the current stack pointer—for the third argument of `getaddrinfo`.

For each argument to the API call, it calculates the address of the corresponding stack location, and passes it to *CollectTypeInformation* (Alg. 6), along with the argument type from the function prototype and the symbolic state. *CollectTypeInformation* is a recursive function that tries to associate each type with the corresponding symbolic expression on the stack. Depending on the type, the actions are slightly different:

Algorithm 6 CollectTypeInformation

Require: A type T

Require: An address *addr*

Require: A symbolic state S

Ensure: Updated database

```
1: if T is a pointer type T'* then
2:     Let sym_expr be the symbolic expression obtained by looking up addr in S.
3:     Insert the mapping (sym_expr, T'*) into the database
4:     Let addr' be the symbolic expression at address sym_expr in S
5:     if addr' is a scalar then
6:         CollectTypeInformation(T', addr')
7:     end if
8: else if T is a basetype then
9:     Let sym_expr be the symbolic expression obtained by looking up addr in S.
10:    Insert the mapping (sym_expr, T) into the database
11: else if T is a structure type then
12:    for all T_i a field type of T do
13:        CollectTypeInformation(T_i, addr + offset_i)
14:    end for
15: end if
```

- In the case of a pointer type $T*$, BCE first adds the mapping (sym_expr, $T*$) to the database, and looks up the corresponding value in the symbolic state, and recursively calls *CollectTypeInformation*, passing the value along with the type T of the object referred to. For example, *CollectTypeInformation*(`ADDRINFO*`, sp) recursively calls

  CollectTypeInformation(`ADDRINFO`, S(*sp*)[13])

- In the case of a basetype T, BCE looks up the corresponding value (sym_expr) in the symbolic state, and it adds the mapping (sym_expr, T). For example, the first token of the

[13] S(*sp*) denotes a lookup of *sp* in symbolic state S.

```
[1] int getaddrinfo (
[2]     char*         nodename;
[3]     char*         servname;
[4]     ADDRINFO*     hints;
[5]     ADDRINFO*     res;
[6] };
[7] struct {
[8]     ....
[9]     char*         ai_canonname;
[10]    sockaddr_in*  ai_addr;
[11]    ....
[12]} ADDRINFO;
[13]struct {
[14]    ....
[15]    unsigned long  sin_zero;
[16]} sockaddr_in;
```

(a)

```
[1] sockaddr_in* s = ...; // malloc
[2] s->sin_zero = atoi(cmd_token[0]);
[3] ADDRINFO* a = ...; // malloc
[4] a->ai_canonname = ...; // malloc
[5] strcpy(a->ai_canonname, cmd_token[1]);
[6] a->ai_addr = s;
[7] getaddrinfo(..., ..., a, ...);
```

(b)

Figure 5.14 (a) The prototypes of `getaddrinfo`, `ADDRINFO`, and `sockaddr_in`; (b) an example code fragment.

command is used for the field `sin_zero` of `sockaddr_in` in Fig. 5.14, which is of base-type `unsigned long`. In this case, BCE collects the information that the associated symbolic expression is of type `unsigned long`.

- In the case of a structure type, such as `struct` or `class`, BCE iterates over the structure's fields, calling *CollectTypeInformation* with each type and the address of the corresponding field. For example, *CollectTypeInformation*(`ADDRINFO`, $S(sp)$) recursively calls *CollectTypeInformation*(`char*`, $S(sp)$ + offset_1), *CollectTypeInformation*(`sockaddr_in*`, $S(sp)$ + offset_2), and so forth, where offset_i is the corresponding offset for each field.

5.2.6 Implementation

The BCE implementation has been structured so that it can be retargeted to different languages easily. The core components of the system are language-independent in two different dimensions:

1. The BCE driver implements Alg. 2. It is structured so that one only needs to provide an implementation of concrete execution and symbolic execution of a language. Consequently, this component of the system can be used for source-level languages or for machine-code languages.
2. For machine-code languages, we used the TSL-generated primitives for concrete execution and symbolic execution. The TSL-generated symbolic-analysis primitives enable to obtain accurate path constraints. Consequently, unlike SAGE or other tools that use approximation—e.g., all non-linear operations (such as multiplication, division, and bitwise arithmetic) as well as symbolic dereferences of pointers, are concretized either for efficiency or due to technical difficulty—BCE guarantees no *divergences* as discussed in §4.7.

Control-Dependence Information. The control-dependence information used for the systematic path-exploration of BCE is collected from the control-dependence graph for a bot program. BCE uses CodeSurfer/x86 [44] to obtain the control-dependence graph for a bot program.

API Call Prototypes. BCE uses IDApro [18] and its Fast Library Identification and Recognition Technology (FLIRT) [9] to identify calls to library functions. It then uses a database of function prototypes and OS and network-related types to extract type information from the recovered symbolic information, as described in §5.2.5.

Library Functions. In BCE, each library-function call is replaced with a simplified model on which concrete and symbolic execution are performed as with other user functions.

5.2.7 Experiments

We performed experiments on four bot programs. The bots are from different families, and they have different sets of commands. Fig. 5.15 summarizes the experimental results. The table

Bot Program			Results				Time						
Name	# Instrs.	% Nf/Nt	# Traces	# SymExprs	# Iterations	Trace Leng	Avg.CE	Total.CE	Avg.SE	Total.SE	Avg.PE	Total.PE	Total
dBot	32168	19%	18	7	89	1893	2.6	231.4	4.8	427.3	0.9	831.3	1489.9
AgoBot	54641	36%	17	8	123	4167	7.9	979.1	12.5	1538.7	16.8	2067.6	4585.4
SpyBot	8360	40%	31	10	279	1290	3.9	1074.2	7.2	2003.2	8.5	2374.3	5451.7
EvilBot	2917	29%	17	4	133	2476	2.5	333.8	4.4	589.2	2.5	328.5	1251.5

Figure 5.15 BCE experiments. The columns, in order, are: the number of instructions (#Instrs); the percentage of nodes marked as either N_f or N_t in the final trace tree; the number of unique traces ending with at least one API call; the number of commands for which BCE provides symbolic expressions; the total number of iterations to identify the traces; the average trace length; the average time taken for concrete execution; the total time taken for concrete execution; the average time taken for symbolic execution; the total time taken for symbolic execution; the average time taken for path exploration; the total time taken for path exploration; and the total time taken in seconds. The experiments were run on a Intel P4 1.79GHz machine with 1.49GB RAM.

Bot Program	Configuration			
Name	w/ CDI & w/ Pruning	w/o CDI & w/ Pruning	w/ CDI & w/o Pruning	w/o CDI & w/o Pruning
dBot	18/89 (20%)	18/101+ (<18%)	18/99+ (<18%)	11/142+ (<8%)
AgoBot	17/123 (14%)	17/172+ (<10%)	17/158+ (<11%)	13/167+ (<8%)
SpyBot	31/279 (11%)	28/281+ (<10%)	27/420+ (<6%)	25/528+ (<5%)
EvilBot	17/133 (13%)	14/206+ (<7%)	17/163+ (<10%)	11/308+ (<4%)

Figure 5.16 BCE experiments. The table reports results for four configurations of BCE: (1) "w/ CDI" and "w/ Pruning", (2) "w/o CDI" and "w/ Pruning", (3) "w/ CDI" and "w/o Pruning", and (4) "w/o CDI" and "w/o Pruning". The numbers reported in each column are the number of unique traces ending with API call(s), the total number of iterations, and the percentage of iterations that resulted in a trace ending with API calls. The experiments were run on a Intel P4 1.79GHz machine with 1.49GB RAM; the symbol "+" after the number of iterations means that BCE with the configuration did not finish (i.e., program exploration could continue infinitely even if all possible commands had been identified.)

first shows the size of each program in terms of the number of instructions, and the percentage of the branches marked as N_f or N_t for each program.

The four columns listed under "Results" shows the number of traces ending with at least one API call, the total number of iterations performed by BCE,[14] and the number of the command

[14] An *iteration* means one run of the basic search step of the BCE algorithm (Alg. 2); on each iteration, a new path is found that leads to a new concrete state.

strings that expect one or more arguments. BCE provides a symbolic expression for such arguments, as discussed in §5.2.5.

For dBot and AgoBot, we had source code and we were able to compare the extracted commands with the commands that one can obtain from the source code. In case of AgoBot, there are two commands—"bot.quit" and "bot.die"—that were not identified as bot commands by BCE, but are actually commands. This is because they are not involved with any Windows API call. Those commands modify some values to change the state of the bot. Even though BCE was able to identify those strings, BCE did not mark them as commands because BCE requires some API call to be controlled by an input string for the string to be classified as a command. Each complete command string, such as "bot.die\0", is extracted through multiple BCE iterations as follows:

"bot.d"

"bot.di"

"bot.die"

"bot.die\0"

If there is no indication that the extracted string is a command (i.e., it controls no API calls), such as "bot.die", there needs to be some manual interpretation of BCE's results, such as whether one should consider an array of bytes in the input that ends with a delimiter (e.g., \0 in case of `strcmp`) to be a command.

We also performed an experiment to determine how well the two state-space-exploration strategies that we introduced in §5.2.3.2 and 5.2.3.3 perform: one strategy chooses a path that has the possibility of encountering API calls (denoted as "w/ CDI"); the other stops further exploration along the current path once the trace encounters an API call (denoted as "w/ Pruning").

The results are shown in Fig. 5.16. We compared the number of traces ending with API calls and the total number of iterations under the configuration "w/ CDI" and "w/ Pruning" with three other configurations—(i) "w/o CDI" and "w/ Pruning", (ii) "w/ CDI" and "w/o Pruning", and (iii) "w/o CDI" and "w/o Pruning". BCE performs best using the configuration "w/ CDI" and "w/ Pruning".

One other way in which the four configurations differed is in their ability to report whether all commands had been found. Only the configuration "w/ CDI" and "w/ Pruning" is able to do this; i.e., it exhausted its (pruned) search space and hence could report that there was nothing more to be found. With the other configurations, BCE did not finish even if it had identified all the commands.

As explained in §5.2.3.3, the user must bear in mind that the commands identified are really command fragments, and various combinations of the command fragments must be tried.

5.2.8 Limitations

BCE currently has the following limitations:

1. *Plain (unpacked) binaries.* BCE only handles unpacked binaries. In principle, directed test generation is applicable even for packed binaries by invoking a decoder on the fly during concrete execution. However, the current implementation of BCE needs a preprocessing step to obtain control-dependence information, which our implementation obtains from a pre-built control-flow graph. One would need some heuristics other than control-dependence information as an alternative for avoiding combinatorial explosion.
2. *Manual identification of the right starting point.* BCE starts its exploration from some command-processing function other than main. This allows relatively short traces for both concrete execution and symbolic execution, resulting in better overall performance of BCE. Typically, there is some initialization code between the beginning of the main function and the command-processing function that is not relevant to extracting input commands. However, this can be problematic if the initialization code affects concrete execution in significant ways. Finding a way to start BCE from the very beginning of a program with low cost is left for future work.
3. *Approximation.* BCE currently approximates some library function calls by using some simplified models. For example, dBot uses `snprintf` as follows to generate a string in a specific format for the purpose of sending a log to the bot-master.

   ```
   snprintf(buf, sizeof(buf), ``%s %s\r\n'', ..., a[x+1]);
   ```

   where `a[x+1]` is one of the command tokens.

A portion of the command is copied into `buf` in `snprintf`. The `buf` is then passed as a parameter to an API call.

If concrete execution and symbolic execution go inside of `snprintf`, BCE can obtain a symbolic expression for `buf` that contains symbols from the input command. Instead of doing that, to simplify BCE's handling of calls to `snprintf`, we model `snprintf` as a copy operator so that the input command symbol `a[x+1]` is copied into the buffer `buf` ignoring the format string.

4. Obfuscation on branch conditions. BCE relies on branch conditions to explore a program. Therefore, if the branch conditions are obfuscated by encryption, it prevents BCE from exploring program paths correctly. For example, fragment (a) below is a normal branch condition that checks a byte value against a constant. As proposed by Sharif et al. [168], the code can be obfuscated as shown in fragment (b). Because it is difficult to invert the hash function, it is infeasible to find `c` given H_c.

```
[1] if (X == c) {
[2]     B
[3] }
```

(a)

```
[1] if (Hash(X) == Hc) {
[2]     run Decrypt(B_E, c)
[3] }
[4] // where Hc = Hash(c), B_E = Encrypt(B, c)
```

(b)

5.2.9 Related Work

Machine-Code Analyzers Targeted at Finding Vulnerabilities. §5.1.5 discussed some work on techniques to detect security vulnerabilities by analyzing source code (for a variety of languages). MineSweeper [55], the work of Moser et al. [139], and SAGE have been discussed in §5.2.3.

Dynamic Techniques. J. Caballero et al. proposed techniques that can be used to extract the format of the protocol messages sent from a bot-master by analyzing bot binaries [58]. They introduced a technique called *buffer deconstruction* that builds the message field tree of a sent message by analyzing how the output buffer is constructed. Furthermore, they used type-inference-based techniques to find out the type information of each field of the extracted structure by monitoring

how the received (or sent) data is used at places where the types are known, such as system calls. Their technique focuses on extracting message formats given proper inputs that trigger malicious actions, whereas BCE aims to extract such proper inputs.

Cho et al. proposed a technique for inferring protocol state machines and applied it to the analysis of botnet Command and Control (C&C) protocols [65]. The inferred protocol state machines can be used for formal analysis for botnet defense, including finding the weakest links in a protocol, uncovering protocol design flaws, inferring the existence of unobservable communication back-channels among botnet servers, etc.

5.2.10 Conclusion

We developed a tool called BCE that automatically extracts botnet-command information from bot executables, without using source code or symbol-table/debugging information. The information obtained using BCE can be used to build up proper input commands that trigger API-level behaviors. BCE furnishes other kinds of information about a bot's commands, in particular, information that combines the recovered symbolic information about inputs with type information for the target API calls. BCE also provides a sequence of API calls controlled by each command, which helps users to understand the bot's API-level behaviors.

BCE performs directed test generation on executables and incorporates a new search technique based on control-dependence information. Our experiments showed that the new search strategies developed for BCE yielded both substantially higher coverage of the parts of the program relevant to identifying bot commands, as well as lowered run-time.

Chapter 6

Conclusion

As discussed in Chapter 2, the problem of analyzing executables to recover information about their execution properties has been receiving increased attention, in part because of the WYSINWYX phenomenon. The WYSINWYX phenomenon is due to several drawbacks of source-code analysis and can be addressed only by machine-code level analysis. The approach of working with machine-code exposes the actual instructions that will be executed, and thus works on an artifact that reveals the actual behavior that arises during program execution.

Although establishing execution properties at the machine-code level is a daunting task due to the challenges of machine-code analysis, as discussed in Chapter 2, several research efforts have been made to develop tools and techniques for machine-code analysis. One major effort is CodeSurfer/x86, of which I was partly involved in the development. In Chapter 2, we presented the two applications that I developed—FFE/x86 and ConSeq—that made use of CodeSurfer/x86.

Unfortunately, although the techniques incorporated into CodeSurfer/x86 are, in principle, language-independent, they were instantiated only for a single instruction set (Intel x86). As already mentioned in Chapter 1, this situation is common in work on program analysis: although the techniques described in the literature are language-independent, analysis implementations are often tied to particular language-specific compiler infrastructure. Unlike the situation in source-code analysis, which can be addressed by developing common intermediate languages, machine-code analysis suffers from the fact that instruction sets typically have hundreds of instructions and a variety of architecture-specific features that are incompatible with other architectures. With future computing platforms based on multicore architectures and transactional memory, future runtime

environments using just-in-time compiling, future systems providing cloud computing and autonomic computing, plus cell phones, PDAs, wearable computers, and autonomous vehicles all entering the fray, both (i) security and reliability problems, and (ii) the variety of computing platforms to analyze will only increase.

To address these concerns, we developed improved techniques for analyzing machine code—in particular, a language called TSL (for "**T**ransformer **S**pecification **L**anguage") for describing the semantics of an instruction set, along with a runtime system to support the creation of a multiplicity of static-analysis, dynamic-analysis, and symbolic-analysis components.

In addition to the two applications to CodeSurfer/x86 presented in Chapter 2, the main contributions that this dissertation made can be summarized as follows:

- In Chapter 3, we presented the TSL system in detail. In the TSL system, analysis components are generated from formal specifications of the abstract syntax and the concrete semantics of an instruction set. TSL was presented from two perspectives: (i) how to write a TSL specification from the point of view of instruction-set-specification developers, and (ii) how to write TSL reinterpretations from the point of view of analysis developers.

  In §3.2, we presented various techniques incorporated to implement the TSL compiler, which translates a specification to a common intermediate representation (CIR). The technical contributions that we made in the design and development of the TSL system can be summarized as follows:

  - *Two-level semantics (along with binding-time analysis):* A two-level CIR allows the precision of an abstract transformer to sometimes be improved—and never made worse—by interpreting subexpressions associated with the manipulation of concrete values in concrete semantics, which the specification of an instruction set often contains. This is done by separating the subexpressions associated with the manipulation of *abstract* values in abstract semantics from other manipulations that can always be treated as *concrete values*. To this end, we made use of the existing technique of binding-time analysis [109].

- *Paired-semantics:* The TSL system allows easy instantiations of *reduced product*s [74] by means of *paired semantics*. One can use the paired-semantics mechanism to obtain desired *multi-phase interactions* among TSL-generated analyzers. By creating a duplicated, but improved CodeSurfer/x86, we demonstrated that this method of CIR instantiation is useful for performing a form of reduced product when analyses are split into multiple phases, as in a tool like CodeSurfer/x86.
- *With-normalization and pattern compilation:* TSL provides a mechanism for deconstruction by means of pattern matching. The TSL front-end performs *with-normalization*, which transforms all multi-level with expressions to use only one-level patterns; an efficient pattern matcher is then generated via the pattern-compilation algorithm developed by Pettersson [153, 178].
- *Execution over abstract states:* An appropriate translation of conditional expressions and recursion functions allows to create abstract interpreters for an instruction-set specification: in particular, the code generated for each transformer is able to: (i) execute over abstract states (§3.2.2), (ii) possibly propagate abstract states to more than one successor in a conditional expression (§3.2.2.1), (iii) compare abstract states and terminate abstract execution when a fixed point is reached (§3.2.2.2), and (iv) apply widening operators, if necessary, to ensure termination (§3.2.2.2).

In chapter 3, we summarized the applications that the TSL system has been applied to, including the various static-analysis components generated from the TSL specification of the IA32 instruction set to develop a new incarnation of CodeSurfer/x86—a revised version whose analysis components are implemented via TSL. The analogous components for the PowerPC32 instruction set were generated from a TSL specification of PowerPC32.

We also discussed the leverage that the TSL system provides in §3.4. We showed that the TSL system provides considerable leverage for implementing analysis tools and experimenting with new ones. New analyses are easily implemented because a clean interface is provided for defining an interpretation.

The reinterpretation mechanism allows TSL to be used to implement *tool-component generators* and *tool generators*. Each implementation of an analysis component's driver (e.g., fixed-point-finding solver, symbolic executor) serves as the unchanging driver for use in different instantiations of the analysis component for different instruction sets. The TSL language becomes the specification language for retargeting that analysis component for different instruction sets.

Furthermore, for a system like CodeSurfer/x86—which uses multiple analysis phases—automating the process of creating abstract transformers ensures *semantic consistency*; that is, because analysis implementations are generated from a *single* specification of the instruction set's concrete semantics, this guarantees that a *consistent* view of the concrete semantics is adopted by all of the analysis implementations used in the system.

- In Chapter 4, we presented a novel way to obtain semantic reinterpretation automatically, via mutually-consistent, correct-by-construction implementations of symbolic primitives—in particular, quantifier-free, first-order-logic formulas for
  - (a) symbolic evaluation of a single command,
  - (b) $\mathcal{WLP}$ with respect to a single command, and
  - (c) symbolic composition for a class of formulas that express state transformations,

  for *every* instruction set for which one has a TSL specification. We also demonstrated that semantic reinterpretation could create such primitives for languages with pointers, aliasing, dereferencing, and address arithmetic.

  As far as we are aware, the application of semantic reinterpretation to a logic is a new idea. A related innovation on which our results rest was to define a particular form of state-transformation formula (structure-update expressions) as a first-class notion in the logic. By this device, such formulas could (i) serve as a replacement domain in the reinterpretations of both the programming language's meaning functions and the logic's meaning functions, and (ii) be reinterpreted themselves.

- In Chapter 5, we presented two applications—MCVETO and BCE—developed using TSL-generated analysis components, which use logic-based search procedures to establish properties of machine-code programs. Compared to work by others on logic-based search procedures for machine code, what distinguishes the work on MCVETO and BCE is that both applications are *goal-directed*. That is, they both have a target property or program point of interest, and this target is used to focus the search.
  - *MCVETO.* MCVETO is a tool to check whether a stripped machine-code program satisfies a safety property. The chapter described how verification of machine code in MCVETO is performed, and discussed how MCVETO avoids using conventional techniques on software model checking that would be unsound if applied at the machine-code level.

    MCVETO is capable of verifying (or detecting flaws in) self-modifying code (SMC). With SMC there is no fixed association between an address and the instruction at that address, but this is handled automatically by MCVETO's mechanisms for abstraction refinement. To the best of our knowledge, MCVETO is the first model checker to handle SMC.

    In Chapter 5, we also presented a language-independent algorithm to identify the aliasing condition relevant to a property in a given state. Unlike previous techniques, it applies when static names for variables/objects are unavailable.

    We also developed several techniques to enhance the methods used during directed proof generation to elaborate the abstraction in use: the techniques enable exhaustive loop unrolling to be avoided by discovering the right loop invariant. The method in which we exploit program invariants allows *soundness to be retained* at all times even though the techniques we use for obtaining invariants are speculative.
  - *BCE.* BCE is a tool for automatically extracting botnet-command information from bot executables, without using source code or symbol-table/debugging information. The information obtained using BCE can be used to build up proper input commands that trigger API-level behaviors. What distinguishes BCE from other existing

symbolic-execution-based test-generation tools is that BCE is goal-directed, using a new search technique that I developed based on control-dependence information.

LIST OF REFERENCES

[1] *1994 Scotland RAF Chinook crash*.
"http://en.wikipedia.org/wiki/1994_Scotland_RAF_Chinook_crash".

[2] *2007 Malware Report*.
"http://www.computereconomics.com/page.cfm?name=Malware%20Report".

[3] *Ariane 5 Flight 501*.
"http://en.wikipedia.org/wiki/Ariane_5_Flight_501".

[4] *CodeSonar, GrammaTech, Inc.*
"http://www.grammatech.com/products/codesonar".

[5] *CodeSurfer, GrammaTech, Inc.*
"http://www.grammatech.com/products/codesurfer".

[6] *compress95*, spec benchmark.
"http://www.itee.uq.edu.au/∼emmerik/specbench.html".

[7] *Coverity Prevent.*
"http://www.coverity.com/products/prevent_analysis_engine.html".

[8] *cpio, GNU* project.
"http://www.gnu.org/software/cpio/cpio.html".

[9] *Fast Library Identification and Recognition Technology, DataRescue sa/nv, Liège, Belgium*.
"http://www.datarescue.com/idabase/flirt.htm".

[10] *File Format Reversing - EverQuest II VPK*.
"http://www.openrce.org/articles/full_vew/16".

[11] *flex*.
"http://www.gnu.org/software/flex/".

[12] *GCC, the GNU Compiler Collection.*
"http://gcc.gnu.org".

[13] *GrammaTech, Inc.*
“http://www.grammatech.com”.

[14] *GZIP file format specification version 4.3*.
“http://www.gzip.org/zlib/rfc-gzip.html”.

[15] *gzip*, gnu project.
“http://www.gzip.org/”.

[16] *Hierarchical State Machine*.
“http://www.eventhelix.com/RealtimeMantra/HierarchicalStateMachine.htm”.

[17] *IA-32 Intel Architecture Software Developer’s Manual*.
“http://developer.intel.com/design/pentiumii/manuals/243191.htm”.

[18] *IDAPro disassembler*.
“http://www.datarescue.com/idabase/”.

[19] *ping*.
“http://packages.debian.org/stable/net/netkit-ping”.

[20] *png2ico*.
“http://www.winterdrache.de/freeware/png2ico/”.

[21] *PPL: The Parma Polyhedra Library*.
“http://www.cs.unipr.it/ppl/”.

[22] *SANS sees upsurge in zero-day Web-based attacks*.
“http://www.computerworld.com/s/article/9005117/SANS_sees_upsurge_in_zero_day_Web_based_attacks”.

[23] *Soot: A Java optimization framework*.
“http://www.sable.mcgill.ca/soot/”.

[24] *tar, GNU* project.
“http://www.gnu.org/software/tar/tar.html”.

[25] *The botnet world is booming*.
“http://www.networkworld.com/news/2009/070909-botnets-increasing.html”.

[26] *The LLVM Compiler Infrastructure*.
“http://www.llvm.org”.

[27] *The PowerPC User Instruction Set Architecture*.
“http://doi.ieeecs.org/10.1109/MM.1994.363069”.

[28] *TVLA System*.
"http://www.cs.tau.ac.il/∼tvla/".

[29] *US Code: Title 17, Sect. 1201(f)*.
"http://www.law.cornell.edu/uscode/html/uscode17/usc_sec_17_00001201—-000-.html#f".

[30] *WALA*.
"http://wala.sourceforge.net/wiki/index.php/".

[31] Intel 64 and ia-32 architectures software developer's manual, volume 2a: Instruction set reference, a-m. "http://download.intel.com/design/processor/manuals/253666.pdf".

[32] Intel 64 and ia-32 architectures software developer's manual, volume 2b: Instruction set reference, n-z. "http://download.intel.com/design/processor/manuals/253667.pdf".

[33] C. Allauzen, M. Riley, J. Schalkwyk, W. Skut, and M. Mohri. OpenFst: A general and efficient weighted finite-state transducer library. In *Proc. 9th Int. Conf. on Implementation and Application of Automata*, 2007.

[34] R. Alur, M. Benedikt, K. Etessami, P. Godefroid, T. Reps, and M. Yannakakis. Analysis of recursive state machines. *TOPLAS*, 27(4), 2005.

[35] R. Alur and M. Yannakakis. Model checking of hierarchical state machines. In *Foundations of Softw. Eng.*, volume 23, 6 of *Softw. Eng. Notes*, pages 175–188, New York, November 3–5 1998. ACM Press.

[36] W. Amme, P. Braun, E. Zehendner, and F. Thomasset. Data dependence analysis of assembly code. *IJPP*, 2000.

[37] K. Ashcraft and D. Engler. Using programmer-written compiler extensions to catch security holes. In *In IEEE Symposium on Security and Privacy*, pages 143–159, 2002.

[38] G. Balakrishnan. *WYSINWYX: What You See Is Not What You eXecute*. PhD thesis, C.S. Dept., Univ. of Wisconsin, Madison, WI, August 2007. Tech. Rep. 1603.

[39] G. Balakrishnan, R. Gruian, T. Reps, and T. Teitelbaum. Codesurfer/x86 – A platform for analyzing x86 executables. In *Comp. Construct.*, 2005.

[40] G. Balakrishnan and T. Reps. WYSINWYX: What You See Is Not What You eXecute. *Trans. on Prog. Lang. and Syst.* (To appear.).

[41] G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *Comp. Construct.*, pages 5–23, 2004.

[42] G. Balakrishnan and T. Reps. DIVINE: DIscovering Variables IN Executables. In *Verif., Model Checking, and Abs. Interp.*, 2007.

[43] G. Balakrishnan and T. Reps. Analyzing stripped device-driver executables. In *Tools and Algs. for the Construct. and Anal. of Syst.*, 2008.

[44] G. Balakrishnan, T. Reps, N. Kidd, A. Lal, J. Lim, D. Melski, R. Gruian, S. Yong, C.-H. Chen, and T. Teitelbaum. Model checking x86 executables with CodeSurfer/x86 and WPDS++. In *CAV*, 2005.

[45] G. Balakrishnan, T. Reps, D. Melski, and T. Teitelbaum. WYSINWYX: What You See Is Not What You eXecute. In *VSTTE*, 2007.

[46] T. Ball, R. Majumdar, T. Millstein, and S.K. Rajamani. Automatic predicate abstraction of C programs. In *Prog. Lang. Design and Impl.*, New York, NY, 2001. ACM Press.

[47] T. Ball and S.K. Rajamani. The SLAM toolkit. In *CAV*, 2001.

[48] M. Barnett, B.-Y.E. Chang, R. DeLine, B. Jacobs, and K.R.M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects*, 2005.

[49] N.E. Beckman, A.V. Nori, S.K. Rajamani, and R.J. Simmons. Proofs from tests. In *Int. Symp. on Softw. Testing and Analysis*, 2008.

[50] D. Beyer, T.A. Henzinger, R. Majumdar, and A. Rybalchenko. Path invariants. In *Prog. Lang. Design and Impl.*, 2007.

[51] L. Birkedal and M. Welinder. Hand-writing program generator generators. In *Prog. Lang. Impl. and Logic Prog.*, 1994.

[52] M. Bishop and M. Dilger. Checking for race conditions in file accesses, 1996.

[53] F. Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Int. Conf. on Formal Methods in Prog. and their Appl.*, LNCS. Springer-Verlag, 1993.

[54] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, P. Poosankam, D. Song, and H. Yin. Automatically identifying trigger-based behavior in malware. In *Botnet Analysis and Defense*. Springer, 2008.

[55] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, P. Poosankam, D. Song, and H. Yin. Automatically identifying trigger-based behavior in malware. In *Botnet Detection*. Springer, 2008.

[56] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *ASPLOS*, 2010.

[57] W.R. Bush, J.D. Pincus, and D.J. Sielaff. A static analyzer for finding dynamic programming errors. *Software: Practice and Experience*, 30:775–802, 2000.

[58] J. Caballero, P. Poosankam, C. Kreibich, and D. Song. Bidirectional protocol reverse engineering: Message format extraction and field semantics inference. Tech. rep. 2009-57, EECS, UC-Berkeley, 2009.

[59] H. Cai, Z. Shao, and A. Vaynberg. Certified self-modifying code. In *Prog. Lang. Design and Impl.*, 2007.

[60] S. Chaki, E. Clarke, A. Groce, J. Ouaknine, O. Strichman, and K. Yorav. Efficient verification of sequential and concurrent C programs. *FMSD*, 25(2–3), 2004.

[61] S. Chaki and J. Ivers. Software model checking without source code. In *Proc. of the First NASA Formal Methods Symposium*, 2009.

[62] T.E. Cheatham, Jr., G.H. Holloway, and J.A. Townley. Symbolic evaluation and the analysis of programs. *Trans. on Softw. Eng.*, 5(4):402–417, 1979.

[63] H. Chen and D. Wagner. MOPS: An infrastructure for examining security properties of software. In *Conf. on Comp. and Commun. Sec.*, pages 235–244, November 2002.

[64] B. Chess. Improving computer security using extended static checking. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, pages 160–, Washington, DC, USA, 2002. IEEE Computer Society.

[65] C.Y. Cho, D. Babi ć, E.C. Shin, and D. Song. Inference and analysis of formal models of botnet command and control protocols. In *Proceedings of the 17th ACM conference on Computer and communications security*, CCS '10, pages 426–439, New York, NY, USA, 2010. ACM.

[66] Jong-Deok Choi et al. Efficient and precise datarace detection for multithreaded object-oriented programs. In *PLDI*, 2002.

[67] A. S. Christensen, A. Møller, and M. Schwartzbach. Precise analysis of string expressions. In *10th International Static Analysis Symposium*, 2003.

[68] M. Christodorescu, W.-H. Goh, and N. Kidd. String analysis for x86 binaries. In *Prog. Analysis for Softw. Tools and Eng.*, 2005.

[69] M. Christodorescu, N. Kidd, and W. Goh. String analysis for x86 binaries. In *Workshop on Program Analysis for Software Tools and Engineering*, 2005.

[70] C. Cifuentes and A. Fraboulet. Intraprocedural static slicing of binary executables. In *ICSM*, pages 188–195, 1997.

[71] T. A. Cook, P. D. Franzon, E. A. Harcourt, and T. K. Miller. System-level specification of instruction sets. In *DAC*, 1993.

[72] K.D. Cooper and K. Kennedy. Interprocedural side-effect analysis in linear time. In *Prog. Lang. Design and Impl.*, pages 57–66, 1988.

[73] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *POPL*, 1977.

[74] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *POPL*, 1979.

[75] M. Cova, V. Felmetsger, G. Banks, and G. Vigna. Static detection of vulnerabilities in x86 executables. In *ACSAC*, 2006.

[76] M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *Prog. Lang. Design and Impl.*, pages 57–68, New York, NY, 2002. ACM Press.

[77] J. W. Davidson and C. W. Fraser. Code selection through object code optimization. In *TPLS*, 1984.

[78] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Int. Conf. on Tools and Algs. for the Construction and Analysis of Systems*, 2008.

[79] B. De Sutter, B. De Bus, K. De Bosschere, P. Keyngnaert, and B. Demoen. On the static analysis of indirect control transfers in binaries. In *Par. and Dist. Proc. Tech. and Appl.*, 2000.

[80] S.K. Debray, R. Muth, and M. Weippert. Alias analysis of executable code. In *POPL*, pages 12–24, 1998.

[81] E. Driscoll, A. Burton, and T. Reps. Checking compatibility of a producer and a consumer. TR 1674, UW-Madison, June 2010.

[82] B. Dutertre and L. de Moura. Yices: An SMT solver, 2006. “http://yices.csl.sri.com/”.

[83] E. Eilam. *Reverse Engineering*. Wiley Publishing, Inc., 2005.

[84] M.D. Ernst, J.H. Perkins, P.J. Guo, S. McCamant, C. Pacheco, M.S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *SCP*, 69(1–3), 2007.

[85] Michael Ernst, Adam Czeisler, William G. Griswold, and David Notkin. Quickly detecting relevant program invariants. In *ICSE*, 2000.

[86] J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *Trans. on Prog. Lang. and Syst.*, 3(9):319–349, 1987.

[87] Cormac Flanagan and Stephen N. Freund. FastTrack: efficient and precise dynamic race detection. In *PLDI*, 2009.

[88] J. S. Foster, R. Johnson, J. Kodumal, and A. Aiken. Flow-sensitive type qualifiers. In *PLDI*, pages 1–12. ACM Press, 2002.

[89] V. Ganesh and D.L. Dill. A decision procesure for bit-vectors and arrays. In *Int. Conf. on Computer Aided Verif.*, 2007.

[90] R. Gerth. Formal verification of self modifying code. In Y. Liu and X. Li, editors, *Proc. Int. Conf. for Young Computer Scientists*, pages 305–311, Beijing, China, 1991. Int. Acad. Pub.

[91] J. T. Giffin, S. Jha, and B. P. Miller. Detecting manipulated remote call streams. In *USENIX Security Symposium*, 2002.

[92] J. T. Giffin, S. Jha, and B. P. Miller. Efficient context-sensitive intrusion detection. In *Proc. of the Network and Distributed System Security Symposium*, 2004.

[93] P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. In *In Programming Language Design and Implementation (PLDI)*, 2005.

[94] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Prog. Lang. Design and Impl.*, 2005.

[95] P. Godefroid, M.Y. Levin, and D. Molnar. Automated whitebox fuzz testing. In *Network and Dist. Syst. Security*, 2008.

[96] P. Godefroid, A.V. Nori, S.K. Rajamani, and S.D. Tetali. Compositional may-must program analysis: Unleashing the power of alternation. In *POPL*, 2010.

[97] Weining Gu, Zbigniew Kalbarczyk, Ravishankar K. Iyer, and Zhen-Yu Yang. Characterization of Linux kernel behavior under errors. In *DSN*, 2003.

[98] B.S. Gulavani, T.A. Henzinger, Y. Kannan, A.V. Nori, and S.K. Rajamani. SYNERGY: A new algorithm for property checking. In *Found. of Softw. Eng.*, 2006.

[99] E. Harcourt, J. Mauney, and T. Cook. Functional specification and simulation of instruction set architectures. In *PLC*, 1994.

[100] K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *Softw. Tools for Tech. Transfer*, 2(4), 2000.

[101] M. Heizmann, J. Hoenicke, and A. Podelski. Nested interpolants. In *POPL*, 2010.

[102] T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, pages 58–70, 2002.

[103] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *TOPLAS*, 12(1):26–60, January 1990.

[104] D. Hovemeyer and W. Pugh. Finding bugs is easy. In *OOPSLA*, 2004.

[105] M. Howard, D. LeBlanc, and J. Viega. *19 Deadly Sins of Software Security*. McGraw-Hill/Osborne, 2005.

[106] R. Jhala and R. Majumdar. B2: Software model checking for C, 2009. "http://www.cs.ucla.edu/∼rupak/b2/".

[107] S.C. Johnson. YACC: Yet another compiler-compiler. Technical Report Comp. Sci. Tech. Rep. 32, Bell Laboratories, 1975.

[108] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall International, 1993.

[109] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall International, 1993.

[110] N.D. Jones and A. Mycroft. Data flow analysis of applicative programs using minimal function graphs. In *POPL*, pages 296–306, 1986.

[111] N.D. Jones and F. Nielson. Abstract interpretation: A semantics-based tool for program analysis. In S. Abramsky, D.M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 4, pages 527–636. Oxford Univ. Press, 1995.

[112] P. A. Karger and R. R. Schell. Multics security evaluation: Vulnerability analysis. Technical report, Hanscom AFB, 1974.

[113] D. Kästner. TDL: a hardware description language for retargetable postpass optimizations and analyses. In *GPCE*, 2003.

[114] N. Kidd, A. Lal, and T. Reps. WALi: The Weighted Automaton Library, 2007. "http://www.cs.wisc.edu/wpis/wpds/download.php".

[115] N. Kidd, T. Reps, D. Melski, and A. Lal. WPDS++: A C++ library for Weighted Pushdown Systems, 2004. "http://www.cs.wisc.edu/wpis/wpds++".

[116] A. King and H. Søndergaard. Automatic abstraction for congruences. In *VMCAI*, 2010.

[117] J. Kodumal and A. Aiken. Banshee: A scalable constraint-based analysis toolkit. In *Static Analysis Symp.*, 2005.

[118] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Automating mimicry attacks using static binary analysis. In *USENIX Sec. Symp.*, 2005.

[119] A. Lal, J. Lim, M. Polishchuk, and B. Liblit. Path optimization in programs and its application to debugging. In *15th European Symposium on Programming*, pages 246–263. Springer, 2006.

[120] A. Lal, J. Lim, and T. Reps. McDash: Refinement-based property verification for machine code. TR-1649, Comp. Sci. Dept., Univ. of Wisconsin, Madison, WI, June 2009.

[121] Akash Lal and Thomas Reps. Reducing concurrent analysis under a context bound to sequential analysis. *Form. Methods Syst. Des.*, 2009.

[122] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. In *Proceedings of the 10th conference on USENIX Security Symposium - Volume 10*, SSYM'01, pages 14–14, Berkeley, CA, USA, 2001. USENIX Association.

[123] P. Lee and M. Leone. Optimizing ML with run-time code generation. In *Prog. Lang. Design and Impl.*, 1996.

[124] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *In Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 141–154. ACM Press, 2003.

[125] J. Lim, A. Lal, and T. Reps. Symbolic analysis via semantic reinterpretation. In *Spin Workshop*, 2009.

[126] J. Lim and T. Reps. A system for generating static analyzers for machine instructions. In *Comp. Construct.*, 2008.

[127] J. Lim, T. Reps, and B. Liblit. Extracting output formats from executables. In *Working Conf. on Rev. Eng.*, 2006.

[128] C. Linn and S.K. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *CCS*, 2003.

[129] B. Livshits and M. Lam. Finding security vulnerabilities in Java applications with static analysis. In *USENIX Sec. Symp.*, 2005.

[130] V. B. Livshits and M. S. Lam. Finding security vulnerabilities in java applications with static analysis. In *Proceedings of the 14th conference on USENIX Security Symposium - Volume 14*, pages 18–18, Berkeley, CA, USA, 2005. USENIX Association.

[131] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes – a comprehensive study of real world concurrency bug characteristics. In *ASPLOS*, 2008.

[132] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. AVIO: detecting atomicity violations via access interleaving invariants. In *ASPLOS*, 2006.

[133] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

[134] K. Malmkjær. *Abstract Interpretation of Partial-Evaluation Algorithms*. PhD thesis, Dept. of Comp. and Inf. Sci., Kansas State Univ., Manhattan, Kansas, 1993.

[135] R. Martin, S. Christey, and J. Jarzombek. The case for common flaw enumeration. "http://cwe.mitre.org/documents/case_for_cwes.pdf".

[136] L. Mauborgne and X. Rival. Trace partitioning in abstract interpretation based static analyzers. In *ESOP*, 2005.

[137] P. Mishra, A. Shrivastava, and N. Dutt. Architecture description language: driven software toolkit generation for architectural exploration of programmable SOCs. *TODAES*, 2006.

[138] J.M. Morris. A general axiom of assignment. In M. Broy and G. Schmidt, editors, *Theor. Found. of Program. Methodology, Proc. of the 1981 Marktoberdorf Summer School*, volume 91 of *NATO Adv. Study Insts. Ser. C, Math. and Phys. Sci.*, pages 25–34. Reidel, 1982.

[139] A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *IEEE Symposium on Security and Privacy*, 2007.

[140] P.D. Mosses. A semantic algebra for binding constructs. In *Int. Colloq. on Formalization of Programming Concepts*, 1981.

[141] M. Müller-Olm and H. Seidl. Analysis of modular arithmetic. In *European Symp. on Programming*, 2005.

[142] Madanlal Musuvathi and Shaz Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *PLDI*, 2007.

[143] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Grard Basler, Piramanayagam A Nainar, and Iulian Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *OSDI*, 2008.

[144] A. Mycroft and N.D. Jones. A relational framework for abstract interpretation. In *Programs as Data Objects*, 1985.

[145] G.C. Necula and P. Lee. Safe kernel extensions without run-time checking. In *Op. Syst. Design and Impl.*, 1996.

[146] G. Nelson. A generalization of Dijkstra's calculus. *Trans. on Prog. Lang. and Syst.*, 11(4), 1989.

[147] Robert H. B. Netzer and Barton P. Miller. Improving the accuracy of data race detection. In *PPoPP*, 1991.

[148] F. Nielson. Two-level semantics and abstract interpretation. *Theor. Comp. Sci.*, 69:117–242, 1989.

[149] F. Nielson and H.R. Nielson. *Two-Level Functional Languages*. Cambridge Univ. Press, 1992.

[150] A. Nori. Personal communication, January 2009.

[151] Soyeon Park, Shan Lu, and Yuanyuan Zhou. Ctrigger: Exposing atomicity violation bugs from their finding places. In *ASPLOS*, 2009.

[152] S. Pees, A. Hoffmann, V. Zivojnovic, and H. Meyr. LISA machine description language for cycle-accurate models of programmable DSP architectures. In *DAC*, 1999.

[153] M. Pettersson. A term pattern-match compiler inspired by finite automata theory. In *CC*, 1992.

[154] U. Pleban and P. Lee. High-level semantics. In *Workshop on Mathematical Foundations of Programming Language Semantics*, 1987.

[155] G. Ramalingam, J. Field, and F. Tip. Aggregate structure identification and its application to program analysis. In *Proc. of Programming Language*, page 119.

[156] G. Ramalingam, J. Field, and F. Tip. Aggregate structure identification and its application to program analysis. In *POPL*, 1999.

[157] N. Ramsey and J.W. Davidson. Specifying instructions' semantics using λ-RTL. Unpublished manuscript, 1999.

[158] N. Ramsey and M. F. Ferandez. New jersey machine-code toolkit arch. spec. technical report. Technical report, 1994.

[159] J. Regehr, A. Reid, and K. Webb. Eliminating stack overflow by abstract interpretation. In *ACM Trans. on Embedded Comp. Systs.*, pages 751–778, 2005.

[160] T. Reps, G. Balakrishnan, and J. Lim. Intermediate-representation recovery from low-level code. In *Part. Eval. and Semantics-Based Prog. Manip.*, 2006.

[161] T. Reps, M. Sagiv, and G. Yorsh. Symbolic implementation of the best transformer. In *Verif., Model Checking, and Abs. Interp.*, pages 252–266, 2004.

[162] Leonid Ryzhyk, Peter Chubb, Ihor Kuz, and Gernot Heiser. Dingo: taming device drivers. In *EuroSys*, 2009.

[163] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM TOCS*, 1997.

[164] E.R. Scherpelz, S. Lerner, and C. Chambers. Automatic inference of optimizer flow functions from semantics meanings. In *PLDI*, 2007.

[165] B. Schlich. *Model Checking of Software for Microcontrollers*. PhD thesis, RWTH Aachen University, Germany, 2008.

[166] D.A. Schmidt. *Denotational Semantics*. Allyn and Bacon, Inc., Boston, MA, 1986.

[167] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *Found. of Softw. Eng.*, 2005.

[168] M. Sharif, A. Lanzi, J. Giffin, and W. Lee. Impeding malware analysis using conditional code obfuscation. In *Network and Dist. Syst. Security*. 2008.

[169] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*. Prentice-Hall, 1981.

[170] D. Siewiorek, G. Bell, and A. Newell. *Computer Structures: Principles and Examples*. Springer-Verlag, 1982.

[171] J. Sifakis. A unified approach for studying the properties of transition systems. *Theor. Comp. Sci.*, 18:227–258, 1982.

[172] M. Siff, S. Chandra, T. Ball, K. Kunchithapadam, and T. Reps. Coping with type casts in C. In *Found. of Softw. Eng.*, pages 180–198, 1999.

[173] A. Srivastava, A. Edwards, and H. Vo. Vulcan: Binary transformation in a distributed environment. MSR-TR-2001-50, Microsoft Research, April 2001.

[174] A. Thakur, J. Lim, A. Lal, A. Burton, E. Driscoll, M. Elder, T. Andersen, and T. Reps. Directed proof generation for machine code. In *CAV*, 2010.

[175] A. Thakur, J. Lim, A. Lal, A. Burton, E. Driscoll, M. Elder, T. Andersen, and T. Reps. Directed proof generation for machine code. TR 1669, UW-Madison, April 2010.

[176] K. Thompson. *Reflections on trusting trust*. Commun. ACM, 1984.

[177] Mandana Vaziri, Frank Tip, and Julian Dolby. Associating synchronization constraints with data in an object-oriented language. In *POPL*, 2006.

[178] P. Wadler. Efficient compilation of pattern-matching. *The Impl. of Func. Prog. Lang.*, 1987.

[179] D. Wagner and D. Dean. Intrusion detection via static analysis. In *2001 IEEE Symposium on Security and Privacy*, 2001.

[180] D. Wagner, J. Foster, E. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Network and Dist. Syst. Security*, February 2000.

[181] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *In Network and Distributed System Security Symposium*, pages 3–17, 2000.

[182] D. W. Wall. *Systems for late code modification*. Code Generation - Concepts, Tools, Techniques. Springer-Verlag, 1992.

[183] M. Weiser. Program slicing. In *IEEE Transactions on Software Engineering*, 1984.

[184] J. Whaley, D. Avots, M. Carbin, and M.S. Lam. Using Datalog with Binary Decision Diagrams for program analysis. In *Asian Symp. on Prog. Lang. and Systems*, 2005.

[185] Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *USENIX Sec. Symp.*, 2006.

[186] Y. Xie and A. Aiken. Saturn: A scalable framework for error detection using Boolean satisfiability. *Trans. on Prog. Lang. and Syst.*, 29(3), 2007.

[187] Y. Xie, A. Chou, and D.R. Engler. ARCHER: Using symbolic, path-sensitive analysis to detect memory access errors. In *Found. of Softw. Eng.*, pages 327–336, 2003.

[188] Min Xu, Rastislav Bodík, and Mark D. Hill. A serializability violation detector for shared-memory server programs. In *PLDI*, 2005.

[189] Yuan Yu, Thomas Rodeheffer, and Wei Chen. Racetrack: Efficient detection of data race conditions via adaptive tracking. In *SOSP*, 2005.

[190] J. Zhang, R. Zhao, and J. Pang. Parameter and return-value analysis of binary executables. In *Comp. Softw. and Applications Conf.*, 2007.

[191] W. Zhang, J. Lim, R. Olichandran, J. Scherpelz, G. Jin, S. Lu, and T. Reps. Conseq: Detecting concurrency bugs through sequential errors. In *ASPLOS*, 2011.

Appendix A: User Guide for TSL

Appendix A describes the *Transformer Specification Language* (TSL). It also contains information about how to write a TSL specification of the programming language of interest (which we call the *subject* language). The TSL system is applicable to both source languages and low-level machine code. Machine-code languages are used in the examples and descriptions in this manual.

TSL is a strongly typed, first-order functional language with a datatype-definition mechanism for defining recursive datatypes, plus deconstruction by means of pattern matching. Much of what a TSL user writes an instruction-set specification is similar to writing an interpreter for an instruction set in first-order ML. The user specifies (i) the abstract syntax of an instruction set, by defining the constructors for a (reserved, but user-defined) type instruction, (ii) an execution-state type, by defining type state.

Lexical Matters. An *identifier* is a sequence of letters, digits, or underscore characters, beginning with a letter or an underscore. Upper- and lower-case letters are considered distinct characters. The following identifiers are reserved and may not be used for other purposes.

true, false, with, default, let, in, phylum, MAP, COMMON,
EXPORT, UNIQUEREP, NOWIDEN, DECLARATIONS,
FUNCTIONLIST, EXPORT_FUNCTIONLIST

Blanks, tabs, and newlines in the specification file are ignored except that they serve to delimit tokens. Comments, delimited by // and a newline, may appear after any token.

TSL Specification. Each TSL specification consists of a list of *declarations*, which are split into two parts: a definition of an abstract syntax, given as a set of grammar rules, and a list of functions. A specification is structured as follows:

NAME: *instruction_set_name*
DECLARATIONS {
    production_declarations
}
FUNCTIONLIST {
    function_declarations
}
EXPORT_FUNCTIONLIST {
    exported_function_declarations
}

DECLARATIONS, FUNCTIONLIST, and EXPORT_FUNCTIONLIST blocks can appear in any order. Each part can be repeated in a specification. DECLARATIONS contains definitions of user-defined types (*production_declarations*). FUNCTIONLIST and EXPORT_FUNCTIONLIST contain user-defined functions (*function_declarations*) and an exported-function list (*exported_function_declarations*), respectively. §A.1, §A.2, and §A.4 describe how to write production, function, and exported-function declarations, respectively.

A.1 Type Definitions (DECLARATIONS)

A.1.1 Phyla, Operators, and Terms

The core of a specification for a given language is the definition of the language's abstract syntax, given as a set of grammar rules. The grammar rules are essentially productions of a regular-tree grammar.

The derivation trees derived from nonterminal symbols are known as *terms* and the set of terms derived from a given nonterminal symbol constitute a *phylum*. The grammar should be viewed as a type-definition mechanism in which the nonterminal symbols are type names and each nonterminal symbol, taken as a type name, denotes a set of values known as a phylum. We often refer to nonterminal symbols as phyla, although more precisely they are the names of phyla. Each production derives terms that can be thought of as *n*-ary records. The alternatives

of a given nonterminal give rise to different record variants. Terms are used both (i) as abstract representations of instructions, operands, and other syntactic constructs and (ii) as computational values. Each production has a name, known as an *operator*, that can be used in computational expressions (in different contexts) both as a record constructor and as a selector that discriminates between variants.

The concepts *phylum*, *operator*, and *term* are defined mutually recursively. A *phylum* is a set of terms. A *term* is the result of applying a *k*-ary operator to *k* terms of the appropriate phyla. A *k*-ary operator is a constructor-function mapping *k* terms to a term. Operators are typed.

Productions, nonterminal symbols, and operator names are defined simultaneously in *phylum declarations*.

Example A.1(a). Let us consider a phylum of binary trees, TREE. Associated with TREE are two operators: Leaf (of arity 0), and Node (of arity 2, with parameter phyla TREE and TREE). TREE can be defined inductively as follows:

1) The term Leaf() is in TREE;

2) If t_1 and t_2 are terms in TREE, then the term Node(t_1, t_2) is in TREE;

3) No other terms are in TREE.

Phylum TREE is the infinite collection of terms

```
{
Leaf(),
Node(Leaf(), Leaf()),
Node(Node(Leaf(), Leaf()), Leaf()),
Node(Leaf(), Node(Leaf(), Leaf())),
...
}
```

A.1.2 Basetypes

Fig. A.1 shows the basetypes that TSL provides. There are two categories of primitive base-types: *unparameterized* and *parameterized*. An unparameterized base-type is just a set of terms. For example, BOOL is a phylum consisting of truth values, INT32 is a phylum consisting of 32-bit signed whole numbers, etc. MAP[α, β] is a predefined parameterized phylum, with parameters α and β. Each of the following is an instance of the parameterized phylum MAP:

```
MAP[INT32,INT8]
MAP[INT32,BOOL]
MAP[INT32,MAP[INT8,BOOL]]
```

TSL provides special syntax for denoting the terms of primitive phyla, often referred to as *constants*. For example, the truth values of phylum BOOL are denoted by true and false, the integers in phylum INT8 are denoted by 0d8, 1d8, 2d8, etc. The syntax of these primitive constants is summarized in Fig. A.1.

Phylum	Terms	Constants
BOOL	false, true	false, true
INT64	64-bit signed integers	0d64, 1d64, 2d64, ...
INT32	32-bit signed integers	0d32, 1d32, 2d32, ...
INT16	16-bit signed integers	0d16, 1d16, 2d16, ...
INT8	8-bit signed integers	0d8, 1d8, 2d8, ...
STR	Sequences of characters. All characters except '\000' permitted.	"" "ab...AB...01...!%..." "\n\r\b\t\f\'\"\\" "\001\002\003..."
MAP[α,β]	Maps	*no constants*

Figure A.1 Syntax of constants of primitive phyla.

Some primitive values do not have corresponding constant denotations. For example, there is no TSL constant corresponding to negative one, since –1 is an expression — the negation function applied to positive one.

§A.3 presents the operators of the TSL base-types.

A.1.3 Syntax

A *production declaration* defines a new operator and includes all terms constructible by that operator in a given phylum. The form of a production declaration is

phylum-name : *operator-name* (*phylum*$_1$ <*identifier*$_1$ > $\cdots$ *phylum*$_k$ <*identifier*$_k$ >) ;

The phylum named by *phylum-name* is referred to as the *left-hand-side phylum*. *phylum*$_1$, ..., *phylum*$_k$ are the *parameters* of the operator *operator-name*. A production declares that all terms constructed by applying k-ary operator *operator-name* to argument terms of phyla *phylum*$_1$, ..., *phylum*$_k$ are members of the left-hand-side phylum. An operator may not be associated with more than one phylum. Each parameter is associated with a name. The parameter names *identifier*$_1$ ... *identifier*$_k$ need to be distinctive in an operator.

Example A.1(b). The following code snippet shows an example of a definition of AST syntax rules:

```
DECLARATIONS {
   reg32: EAX() | EBX();
   operand: DirectReg32(reg32<Reg>)
          | Immediate32(INT32<Val>)
          ;
   instruction: ADD(operand<Op1> operand<Op2>)
          | ...
          ;
   state: State(MAP[INT32,INT8]<Memory> MAP[reg32,INT32]<Registers>)
}
```

A.1.4 Reserved, but User-Defined Types

Each instruction-set specification must include definitions of the following types:

reg64, reg32, reg16, reg8, cc, instruction, and state

Exported phyla are treated as interfaces between a TSL specification of a subject language and a client analysis for the language.

Each reserved type is annotated with EXPORT and either <E> or <R> (*binding-directive*). There are two kinds of phyla: *concrete* phyla and *abstract* phyla. If a phylum is only used as a concrete type, such as `reg32` and `instruction`, the phylum is annotated with <E>. If a phylum is to be used in a reinterpreted semantics, such as `state`, the phylum is annotated with <R>. The TSL system generates a common intermediate representation in which phyla annotated with <E> are converted to concrete types, and the ones annotated with <R> support both concrete types and reinterpreted versions of those types.

Example A.1(c). Because `reg32`, `instruction`, and `state` are reserved, the code in Example 2.2.1 (b) is amended as follows:

```
DECLARATIONS {
   EXPORT reg32<E>: EAX() | EBX();
   operand: DirectReg32(reg32<Reg>)
          | Immediate32(INT32<Val>)
          ;
   EXPORT instruction<E>
          : ADD(operand<Op1> operand<Op2>)
          | ...
          ;
   EXPORT state<E>: State(MAP[INT32,INT8]<Memory> MAP[reg32,INT32]<Registers>);
}
```

A.1.5 Redefinable Phylum Definitions

TSL allows one to associate base-types (especially parameterized base-types, such as MAP) with other names. Each phylum defined as reinterpretable can be reinterpreted in a client application. The form of a reinterpretable-type declaration is

phylum *phylum identifier*;
phylum MAP[$phylum_1$ <*binding-directive*>, *phylum_2*] *identifier*;

Binding-directive (<E> | <R>) controls the reinterpretation property of the key type of the map. *binding-directive* <E> is used in the examples in this chapter.[1]

In addition to the unparameterized base-types, such as BOOL and INT32, such user-defined reinterpretable types, such as MEMMAP32_8 and REGMAP32, are reinterpreted with new types provided by an analysis developer to create an analysis component.

Example A.1(d). The following code is a part of file-system definition. FILESTREAM is defined as MAP[INT64<E>,INT8]; the key type of FDATA is renamed as inode; and FDATA is defined as MAP[inode<E>,FILESTREAM].

```
DECLARATIONS {
    phylum MAP[INT64<E>,INT8] FILESTREAM;
    phylum INT8 inode;
    phylum MAP[inode<E>,FILESTREAM] FDATA;
}
```

Example A.1(e). The code in Example 2.1 (c) can be rewritten by replacing MAP[INT32,INT8] and MAP[reg32,INT32] with the redefined names MEMMAP32_8 and REGMAP32, respectively.

[1] Ordinarily, the key types of maps are <E>. <R> is used in a few circumstances, but certain conditions must hold for such a reinterpretation to work correctly. The TSL system does not check whether such a reinterpretation obeys the necessary conditions.

```
DECLARATIONS {
   EXPORT reg32<E>: EAX() | EBX();
   operand: DirectReg32(reg32<Reg>)
          | Immediate32(INT32<Val>)
          ;
   EXPORT instruction<E>
          : ADD(operand<Op1> operand<Op2>)
          | ...
          ;
   phylum MAP[INT32<E>,INT8] MEMMAP32_8;
   phylum MAP[reg32<E>,INT32] REGMAP32;
   EXPORT state<E>: State(MEMMAP32_8<Memory> REGMAP32<Registers>);
}
```

A.1.6 Phylum Directives

TSL provides two optional directives–COMMON and UNIQUEREP–for phylum declarations.

- COMMON directive. A phylum can be shared among various languages by annotating the phylum declarations with the directive COMMON. For example, the phylum definitions for modeling context-switches are language-independent.

```
COMMON phylum MAP[reg32<E>,INT32] SAVEREGS;
COMMON phylum MAP[cc<E>,BOOL] SAVEFLAGS;
COMMON context :  Context(SAVEREGS<SaveRegs> SAVEFLAGS<SaveFlags>);
```

- UNIQUEREP directive. A phylum prefixed with UNIQUEREP is translated into a type that only allows a single instance to be constructed of any given term. For example, if `QFBVFormula` is annotated with UNIQUEREP, there is only one instance for each term of `QFBVFormula`, such as `QFBV_TRUE()` and `QFBV_LT(QFBVSymbol32("Sym1"), QFBVScalar32(0))`.

```
COMMON UNIQUEREP QFBVFormula
   :  QFBV_TRUE() | QFBV_FALSE()
   |  QFBV_LT(QFBVTerm32<t1> QFBVTerm32<t2>)
   |  QFBV_AND(QFBVFormula<f1> QFBVFormula<f2>)
   |  QFBV_OR(QFBVFormula<f1> QFBVFormula<f2>)
   |  ...
   ;
```

UNIQUEREP cannot precede COMMON.

A.2 Function Definitions (FUNCTIONLIST)

The form of a *function declaration* is

[*directives*] *phylum*$_0$ *function-name* (
phylum$_1$ *parameter-name*$_1$,
phylum$_2$ *parameter-name*$_2$,
...,
phylum$_k$ *parameter-name*$_k$
) { *expression* } ;

It declares *function-name* to be a *k*-ary function with result phylum *phylum*$_0$, and has, for each *i*, $1 \leq i \leq k$, a parameter named *parameter-name*$_i$ of type *phylum*$_i$. The body of the function, *expression*, is an expression over *parameter-name*$_1$, ..., *parameter-name*$_k$ that must evaluate to a term in the result phylum *phylum*$_0$.

Function declarations are global — they cannot be defined inside one another, nor can they be defined within the scope of productions. Functions are not first-class objects, i.e., they cannot be the value of a parameter or an expression. Functions can be recursive.

A.2.1 Function Directives

This section contains information about function directives, which direct how a function is translated into the common-intermediate representation. Function directives direct the way the TSL system translates a function. TSL supports the following directives for function definitions:

COMMON, NOWIDEN, and CACHED

- COMMON. A function can be shared among various languages by annotating the function declaration with the directive COMMON. The directive COMMON causes the function to be generated in a common namespace. This directive can be only used for functions that are language/instruction-set-indedepndent. E.g.,

```
COMMON INT32  isSignedOverflowForAddition(INT32 a, INT32 b) {
  ....
};
```

- CACHED. The directive CACHED causes TSL to implement function-caching for the function. E.g.,

```
CACHED BOOL  Eval_Formula(Formula f, state S) {
     // expression
};
```

  For example, the return values of the function Eval_Formula for each actual argument pair <f, S> are cached so that they can be retrieved the next time the function is called with the same pair of actuals, instead of evaluating the whole function again.

- NOWIDEN. When a tail-recursive function has a reinterpretable argument type or reinterpretable return type, the default way of translating the reinterpretable version of the function in the CIR is to create a function template that will invoke a widening operation to ensure termination [126]. The directive NOWIDEN causes the TSL compiler to translate the function to a recursive C++ function that does not perform widening. This directive can be used in the cases when termination is guaranteed even without widening. E.g.,

```
CACHED NOWIDEN BOOL  Eval_Formula(Formula f, state S) {
  Formula f1 = f.Arg1();
  Formula f2 = f.Arg2();
  return Eval_Formula(f1, S) && Eval_Formula(f2, S),
}
```

A.3 Expressions

Expressions occur in function declarations. §A.3.1 discusses variables in expressions. §A.3.2 and §A.3.3 discuss applications of functions and operators, and basetype operators, respectively. §A.3.4 presents conditional and binding expressions.

A.3.1 Variables

A *variable* is a name bound to a *value*. The different lexical contexts of expressions give rise to the distinct sorts of variables itemized below.

Parameters of functions. Each parameter of a function is a variable that denotes the value of the corresponding argument passed to the function. The type of such a variable is the one specified for the parameter in the function declaration.

Pattern variables. Patterns in with-expressions, (described in §A.3.4), contain pattern variables. Pattern matching binds each pattern variable to some term. Each pattern variable *p* has a scope within which *p* is a variable that denotes the term to which it has been bound. The type of a pattern variable *p* is determined by the context in which it first occurs in a pattern. This context is either the *i*-th argument of some operator *g*, in which case the type of *p* is the phylum specified for the *i*-th parameter of *g*, or it is an entire pattern, in which case the type of *p* is the type of the expression against which *p* is being matched.

Let-bound variables. Binding lists of let-in-expressions, as described in §A.3.4, create variables whose scope is the expression that follows the in keyword.

A.3.2 Application of functions and operators.

The application of a *k*-ary function or operator to *k* arguments of the appropriate phyla is an expression.

Function applications. A function application has the form

function-name (*expression*$_1$, . . . , *expression*$_k$)

Assume that *function-name* has been declared by

phylum$_0$ *function-name* (
  phylum$_1$ *parameter-name*$_1$,
  . . . ,
  phylum$_k$ *parameter-name*$_k$
) { *expression* } ;

and further assume that arguments *expression*$_1$,. . ., *expression*$_k$ have values v_1, . . ., v_k, respectively. Then the value of the function application is the value of *expression* evaluated in an environment in which parameters parameter-name$_1$, . . ., parameter-name$_k$ are bound to v_1, . . ., v_k, respectively. The types of *expression*$_1$,. . ., *expression*$_k$ must be *phylum*$_1$, . . ., *phylum*$_k$, respectively. The type of the application is *phylum*$_0$. If *function-name* is nullary, an empty pair of parentheses is still required to indicate function application.

Operator applications. An operator application has the form

operator-name (*expression*$_1$, *expression*$_2$, . . ., *expression*$_k$)

Assume the operator has been declared by

phylum-name
: *operator-name* (*phylum*$_1$ $<name_1>$ *phylum*$_2$<*name*$_2$> $\cdots$ *phylum*$_k$<*name*$_k$>);

and further assume that arguments *expression*$_1$, ..., *expression*$_k$ have values v_1, ..., v_k respectively. Then the value of the operator application is the term *operator-name*(v_1, ..., v_k). The types of *expression*$_1$, ..., *expression*$_k$ must be *phylum*$_1$, ... *phylum*$_k$, respectively. The type of the application is *phylum-name*.

A.3.3 Operations on primitive phyla.

A collection of operations on primitive values is built into TSL. Operations for which special syntax is provided are summarized in Fig. A.2. Library functions on basetypes are summarized in Fig. A.3. The two arguments of a binary expression must be expressions of the same type.

A.3.4 Conditional and binding expressions.

Conditional and binding expressions permit the value of an expression to depend on the value of a constituent subexpression. Three forms are allowed: *with-expression*, *conditional-expression*, and *let-expression*.

With-expressions. A *with-expression* is a multi-branch conditional expression that permits discrimination based on the structure of the value of a given expression. The syntax of a with-expression is

```
with ( identifier ) (
  pattern_1 : expression_1,
  pattern_2 : expression_2,
  ...
  pattern_n : expression_n
)
```

The value of *identifier* is called the *matched value*. The value of the with-expression is the value of the *expression*$_i$ corresponding to the first *pattern*$_i$ that *matches* the matched value. Each *pattern*$_i$ may contain *pattern variables*, which, if the match succeeds, are bound to constituents of the

Result	Syntax	Operation
BOOL	b_1 && b_2	logical conjunction of b_1 and b_2
	b_1 || b_1	logical disjunction of b_1 and b_2
	b_1 ^^ b_1	exclusive logical disjunction of b_1 and b_2
	! *b*	logical negation of b
	random(BOOL)	random boolean value
	e_1 < e_2	e_1 less than e_2
	e_1 <= e_2	e_1 less than or equal to e_2
	e_1 > e_2	e_1 greater than e_2
	e_1 >= e_2	e_1 greater than or equal to e_2
	e_1 <u e_2	e_1 less than (unsigned) e_2
	e_1 <=u e_2	e_1 less than or equal to (unsigned) e_2
	e_1 >u e_2	e_1 greater than (unsigned) e_2
	e_1 >=u e_2	e_1 greater than or equal to (unsigned) e_2
	e_1 == e_2	e_1 equal to e_2
	e_1 != e_2	e_1 not equal to e_2
INT64 INT32 INT16 INT8	i_1 * i_1	product of i_1 and i_2
	i_1 / i_2	quotient of i_1 and i_2
	i_1 + i_2	sum of i_1 and i_2
	i_1 – i_2	difference of i_1 and i_2
	i_1 % i_2	i_1 mod i_2
	i_1 & i_2	bitwise-and of i_1 and i_2
	i_1 ^i_2	bitwise-exclusive-or of i_1 and i_2
	i_1 | i_2	bitwise-inclusive-or of i_1 and i_2
	– i	negation of i
	~ i	bitwise-complement of i
	random(α)	random integer value
MAP[α,β]	[*OPAQUE_TYPE*: $\alpha \mapsto e$]	empty map from α with default value e
	m[e_1|– >e_2]	map m updated so that the image of e_1 is e_2
	random(*OPAQUE_TYPE*)	random map

Figure A.2 Operations on the primitive phyla. (In this table, b's are BOOL parameters, i's are integer parameters, m's are MAP parameters, e's are parameters of arbitrary type, α and β are phyla, and *OPAQUE_TYPE* is a reinterpretable map-type defined in a phylum statement (see §A.1.5).)

matched value. The value of *expression*$_i$ is then computed in terms of those bindings. The types of all *expression*$_i$ must be the same phylum p; the type of the entire with-expression is that phylum p.

Result	Function(parameters)	Operation
INT32	Int8To32ZE(INT8 i)	zero-extension of i to 32-bit value
	Int16To32ZE(INT16 i)	zero-extension of i to 32-bit value
	Int8To32SE(INT8 i)	sign-extension of i to 32-bit value
	Int16To32SE(INT16 i)	sign-extension of i to 32-bit value
	Int64To32(INT64 i)	truncation of i to 32-bit value
	BoolToInt32(BOOL b)	if b is true, return 1, otherwise 0
	unsignedDiv32(INT32 i_1, INT32 i_2)	unsigned division of i_1 by i_2
BOOL	getBit32(INT32 i_1, INT32 i_2)	get the bit value at the index i_2 in the 32-bit value i_1
	signedOverflowAdd32(INT32 i_1, INT32 i_2)	return true if an overflow occurs in a signed addition
	signedOverflowSub32(INT32 i_1, INT32 i_2)	return true if an overflow occurs in a signed subtraction
	unsignedOverflowAdd32(INT32 i_1, INT32 i_2)	return true if an overflow occurs in an unsigned addition
	unsignedOverflowSub32(INT32 i_1, INT32 i_2)	return true if an overflow occurs in an unsigned subtraction
STR	ConcatSTR(STR s_1, STR s_2)	concatenation of s_1 and s_2
	SubSTR(STR s, INT32 i_1, INT32 i_2)	sub-string of s from index i_1 to i_2
	INT32toSTR(INT32 i)	convert 32-bit integer value to a string

MEMMAP32_8	MemAccess_32_8_LE_32(MEMMAP32_8 m, INT32 i)	32-bit little-endian memory access addressed by i
	MemUpdate_32_8_LE_32(MEMMAP32_8 m, INT32 i_1, INT32 i_2)	32-bit little-endian memory update
	MemAccess_32_8_BE_32(MEMMAP32_8 m, INT32 i)	32-bit big-endian memory access addressed by i
	MemUpdate_32_8_BE_32(MEMMAP32_8 m, INT32 i_1, INT32 i_2)	32-bit big-endian memory update

Figure A.3 Library functions on the primitive phyla. In this table, i's are integer parameters, s's are STR parameters, and b's are BOOL parameters; MEMMAP32_8 is a reinterpretable map-type whose original type is MAP[INT32,INT8]; m's are MAP-type parameters.

The patterns of a given with-expression must be exhaustive, i.e., it must be possible for the compiler to determine statically that for every evaluation of the given with-expression, one of the patterns will match. This will always be the case if one of the patterns is * or default.

Patterns are defined inductively, as follows:

1) Constants of primitive phyla (TSL base-types) are patterns.
2) Pattern variables are patterns. A pattern variable is an identifier.
3) Both the symbol * and the keyword default are patterns.
4) A k-ary operator *operator-name* applied to k patterns is a pattern:

operator-name (*pattern*$_1$, ..., *pattern*$_k$)

The same pattern variable may occur multiple times in a pattern. The leftmost occurrence of a given pattern variable is its *binding occurrence* and all subsequent occurrences in the same pattern are *bound occurrences*. The type of a pattern variable p is determined by the context of its binding occurrence. This context is either the i-th argument of some operator g, in which case the type of p is the phylum specified for the i-th parameter of g, or it is an entire pattern, in which case the type of p is the type of the expression against which p is being matched.

Let p be a pattern and t be a term. Then p is said to *match* t under the following circumstances:

1) When p is a constant of a primitive phylum and t is the same constant.
2) When p is the binding occurrence of a pattern variable pv, in which case pv is bound to t.
3) When p is a bound occurrence of a pattern variable pv that has been bound to some term t' and t==t'.
4) When p is either * or default.
5) When p is *op*(p_1, ..., p_k) and t is *op*(t_1, ..., t_k) and p_i matches t_i for all i, $1 \leq i \leq k$.

The lexical scope of a pattern variable bound in some *pattern*$_i$ begins at its binding occurrence and extends through the corresponding *expression*$_i$. The scope of pattern variables is block-structured, i.e., a given pattern variable may be redeclared in an inner scope.

Example A.3.4(a). Consider the following definitions of phyla ENV and BINDING:

```
ENV
  : NullEnv()
  | EnvConcat( BINDING ENV )
  ;
BINDING: Binding( INT32 INT8 );
```

A value env of phylum ENV is analyzed by the with-expression that is the body of function lookup:

```
BINDING lookup(INT32 id, ENV env) {
  with (env) (
    NullEnv():  Binding(7d32, 0d8),
    EnvConcat(b, e):
      with (b) (
        Binding(s, *): id==s ? b : lookup(id, e)
      )
  )
};
```

The two operators NullEnv and EnvConcat exhaust all possible alternatives for ENV, so no default pattern is necessary. If the value of env is NullEnv(), then the pattern NullEnv() matches it and the value of the with-expression is Binding(7d32, 0d8). Otherwise, the value of env is necessarily a pair and the pattern EnvConcat(b, e) matches with pattern variables b and e bound to the first and second components, respectively. In this case, the value of the with-expression is the value of the inner with-expression, wherein pattern variables b and e have types BINDING and ENV, respectively.

The same effect can be obtained by combining the two nested with-expressions into one:

```
BINDING lookup(INT32 id, ENV env) {
  with (env) (
    NullEnv():  Binding(7d32, 0d8),
    EnvConcat(Binding(s, v), e):
      id==s ? Binding(s, v) : lookup(id, e)
  )
};
```

Conditional-expressions. A more traditional form of conditional expression is available in TSL, based not on pattern matching but on the value of a Boolean expression. A *conditional-expression* has the form

expression$_1$? *expression*$_2$: *expression*$_3$

When *expression*$_1$ is an identifier i, it is exactly equivalent to the expression

with (i) (true : *expression*$_2$, false : *expression*$_3$)

Let-expressions. Let-expressions are useful for binding values to names. The simplest form of let-expression is:

let *id* = *expression*$_1$ in (*expression*$_2$)

When several values are to be matched, a more general form is available:

let *id*$_1$ = *expression*$_1$; *id*$_2$ = *expression*$_2$; . . . *id*$_i$ = *expression*$_i$ in (
expression$_0$
)

An occurrence of a variable cannot be rebound in subsequent bindings. The last binding in a let-expression is effective in *expression*$_0$. The type of the let-expression is the type of *expression*$_0$. A semicolon before the keyword in is optional.

The value of the general form of let-expression is determined as follows:

Each identifier is bound to the value of the corresponding *expression*. The value of the let-expression is the value of *expression*$_0$ as computed in an environment containing bindings for all variables. The *expressions* of a binding are all evaluated in the environment with includes all variables bound in all previous patterns up to, or the initial environment in case of the first binding.

A.4 Export Function Definitions (EXPORT_FUNCTIONLIST)

A function can be exported to the interface available to client analyses by using a declaration of the following form:

```
EXPORT <cir-directive> function-name (<cir-directive>, ..., <cir-directive>);
```

The TSL compiler only translates functions derivable from the exported functions.
Each *cir-directive* is either <E> or <R>. The return type and parameter types of an exported function are annotated with either <E> or <R> in a EXPORT_FUNCTIONLIST block. <E> directs the TSL system to translate the type as non-reinterpretable, whereas <R> causes the type to be translated as reinterpretable.

Example A.4(a). The concrete semantics of each instruction is specified by defining the function interpInstr, which takes an instruction and a state, and returns an updated state that captures the semantics of the instruction.

```
FUNCTIONLIST {
   state interpInstr(instruction I, state S) {
     ...
   };
}
```

Example A.4(b). interpInstr in Example 2.4(a) can be translated into two versions of CIRs by including the following *export-function declarations* as follows:

```
EXPORT_FUNCTIONLIST {
   EXPORT <E> interpInstr(<E>, <E>);
   EXPORT <R> interpInstr(<E>, <R>);
   ...
}
```

The first export declaration, in which all the types are declared as <E>, generates a component that can be used for creating an emulator for the subject language. With the second declaration,

in which the state types for both the input and the output are <R>, the interpInstr is to be reinterpreted in an alternative semantics:

```
state# interpInstr(instruction I, state# S) {
  ...
};
```

A.4.1 Reserved, but User-Defined Functions (Exported Functions)

Tab. A.1 shows a list of TSL reserved exported functions.[2] The set of exported functions specifies the interface between a specification and an analysis client to create an analysis component. A specification must contain an EXPORT_FUNCTIONLIST block with an export-function declaration for each of the functions in Tab. A.1.

This section described how to write a concrete semantics of a subject language in TSL from the point of view of instruction-set-specification (ISS) developers. The TSL compiler automatically generates from a TSL specification a *common intermediate representation* (CIR) that can be instantiated to create multiple analysis components. This chapter presents how the TSL system generates the CIR (§A.5), as well as how the CIR is instantiated to create an analysis component (§A.6) from the point of view of analysis developers.

A.5 Common Intermediate Representation

The TSL system automatically generates a CIR from a TSL specification of the concrete operational semantics of an instruction set. Each generated CIR is *specific* to a given instruction-set specification, but *common* (whence the name CIR) across generated analyses. CIR is a template class that takes as input a class BT, an abstract domain for an analysis, as shown in Fig. A.5.[3] This section describes how the IR that TSL uses internally to represent a TSL specification (henceforth called TSL-IR) is translated into the output CIR. A specification in TSL is simply linearized,

[2]The list can vary depending on the client analysis system: the table shows a list of reserved, but user-defined functions for machine-code instruction sets.

[3]CIR is in C++ in reality.

Function Name	Discription
state interpInstr(instruction I, state S)	specifies the concrete operational semantics of instruction I
INT32 GetPC32()	returns the program counter (PC)
INT32 GetSP32()	returns the stack pointer (SP)
INT32 AccessPC(state S)	returns the value of PC in state S
state UpdatePC(state S, INT32 v)	updates the value of PC with v in state S
INT32 AccessSP(state S)	returns the value of SP in state S
state UpdateSP(state S, INT32 v)	updates the value of SP in state S
INT32 GetEA32(instruction I)	returns the PC value of instruction I
INT32 GetInstrSize(instruction I)	returns the size of instruction I
INT32 TopOfState32(state S)	returns value at the address pointed to by SP
state Pop32(state S)	adjusts SP to pop the top value
state Push32(state S, INT32 v)	pushes the value v to SP

Table A.1 Reserved exported functions; a complete list of reserved export functions can be found in

TSL/instruction_sets/common/exports.tsl

in evaluation order, into a series of C++ statements, in which the names of basetypes, basetype-operators, and *access/update* functions are prepended with BT ::. The user-defined abstract syntax (lines 3–16 of Fig. A.4) is translated to a set of C++ abstract-domain classes (lines 2–17 of Fig. A.5) that contain appropriate abstract operators. The user-defined types, such as reg32, operand32, and instruction, are translated to abstract C++ classes, and the constructors, such as `eax`, Indirect32, and `add32_32`, are subclasses of the parent abstract C++ class. Each user-defined function is translated to a CIR member function.

Each TSL basetype and basetype-operator is prepended with the template parameter name BT; BT is supplied for each analysis by an analysis developer. The TSL basetype-operator + on line 42 in Fig. A.4 is translated into a static function call on BT::`Plus`, as shown on line 42 in Fig. A.5.

The with expression and the pattern matching on lines 35–45 of Fig. A.4 are translated to switch statements in C++[4] (lines 35–45 in Fig. A.5).

A.5.1 Translation to Two-Level Common Intermediate Representation

This section describes a mechanism for improving a certain level of precision of analyzers by separating concrete and abstract semantics ($\grave{a}$ la Nielson and Nielson [149]).

The concrete semantics of an instruction set often contains some manipulations of values that should always be treated as concrete values (for every abstract interpretation of the instruction). For example, the ISS developer could follow the approach taken in the PowerPC manual [27] and specify variants of the conditional branch instruction (BC, BCA, BCL, BCLA) of PowerPC by interpreting one of the fields in the instruction to determine which of the four variants is being executed. In this case, the precision of an abstract transformer could be harmed by interpreting such subexpressions in the abstract semantics. For instance, in a TSL expression $v = (b\ ?\ 1 : 2)$, where b is definitely a concrete value, v can get a precise value—either 1 or 2—when b is concretely interpreted. However, if b is not expressible precisely in a given abstract domain, the conditional expression "$(b\ ?\ 1 : 2)$" will be evaluated by joining the two branches and v will not hold a precise value.

To address this issue, we perform a kind of binding-time analysis on the TSL-IR, in which the expressions associated with the manipulation of concrete values in an instruction are annotated with C, and others with A. Then, we generate a *two-level* CIR by appending CONC_SEM for C values, and ABS_SEM for A values. The generated CIR is instantiated for an analysis transformer by defining ABS_SEM. We provide a predefined concrete semantics for CONC_SEM.

A.6 CIR Instantiation

This section describes how an analysis developer instantiates the CIR to create an analysis component.

[4]The TSL front end performs *with-normalization*, which transforms all (multi-level) with expressions to use only one-level patterns, using the pattern-compilation algorithm from [153, 178].

The generated CIR is instantiated for an analysis by defining (in C++) an *interpretation*: a representation class for each TSL basetype, and implementations of each TSL basetype-operator. Tab. A.2 shows the implementations of primitives for three selected analyses: value-set analysis (VSA [41]), quantifier-free bit-vector semantics (QFBV), and def-use analysis (DUA). Each interpretation defines an abstract domain. For example, line 3 of each column defines the abstract-domain class for INT32: ValueSet32, QFBVTerm32, and UseSet. Each abstract domain is also required to contain a set of reserved functions, such as *join*, *meet*, and *widen*, which forms an additional part of the API available to analysis engines that use TSL-generated transformers.

A.6.1 Required Operators of Abstract Domains for a TSL Reinterpretation

Fig. A.6 shows the required operators that an abstract domain must provide for a TSL reinterpretation. An abstract domain for a map-basetype must provide mapAccess and mapUpdate.

A.6.2 Paired-Semantics

Our system allows easy instantiations of *reduced product*s [74] by means of *paired semantics*. The TSL system provides a template for paired semantics as shown in Fig. A.7.

The CIR is instantiated with a *paired* semantic domain defined with two interpretations, INTERP1 and INTERP2 (each of which may itself be a paired semantic domain), as shown on line 1 of Fig. A.8. The communication between interpretations may take place in basetype-operators or *access/update* functions; Fig. A.8 is an example of the latter. The two components of the paired-semantics values are deconstructed on lines 3–6 of Fig. A.8, and the individual INTERP1 and INTERP2 components from *both* inputs can be used (as illustrated by the call to *interact* on line 7 of Fig. A.8) to create the paired-semantics return value, answer. Such overridings of basetype-operators and *access/update* functions are done by C++ explicit specialization of members of class templates (this is specified in C++ by "template<>"; see line 2 of Fig. A.8).

This method of CIR instantiation is also useful to perform a form of reduced product when analyses are split into multiple phases, as in a tool like CodeSurfer/x86. CodeSurfer/x86 carries out many analysis phases, and the application of its sequence of basic analysis phases is itself

Table A.2 Parts of the declarations of the basetypes, basetype-operators, and map-access/update functions for three analyses.

VSA	QFBV	DUA
[1] class VSA_INTERP {	[1] class QFBV_INTERP {	[1] class DUA_INTERP {
[2] // basetypes	[2] // basetype	[2] // basetype
[3] typedef ValueSet32 INT32;	[3] typedef QFBVTerm32 INT32;	[3] typedef UseSet INT32;
[4] ...	[4] ...	[4] ...
[5] // basetype operators	[5] // basetype operators	[5] // basetype operators
[6] INT32 Add(INT32 a, INT32 b)	[6] INT32 Add(INT32 a, INT32 b)	[6] INT32 Add(INT32 a, INT32 b)
[7] {	[7] {	[7] {
[8] return a.addValueSet(b);	[8] return QFBVPlus32(a, b);	[8] return a.Union(b);
[9] }	[9] }	[9] }
[10] ...	[10] ...	[10] ...
[11] // map-basetypes	[11] // map-basetypes	[11] // map-basetypes
[12] typedef Dict<reg32,INT32>	[12] typedef Dict<var32,INT32>	[12] typedef KillUseSet VAR32MAP;
[13] REGMAP32;	[13] VAR32MAP;	[13] ...
[14] ...	[14] ...	[14] // map-basetype operators
[15] // map-basetype operators	[15] // map-basetype operators	[15] INT32 Access(
[16] INT32 Access(	[16] INT32 Access(	[16] REGMAP32 m, reg32 k) {
[17] REGMAP32 m, reg32 k) {	[17] REGMAP32 m, reg32 k) {	[17] return UseSet(k);
[18] return m.Lookup(k);	[18] return m.Lookup(k);	[18] }
[19] }	[19] }	[19] REGMAP32
[20] REGMAP32	[20] REGMAP32	[20] Update(REGMAP32 m,
[21] Update(REGMAP32 m,	[21] Update(REGMAP32 m,	[21] reg32 k, INT32 v) {
[22] reg32 k, INT32 v) {	[22] reg32 k, INT32 v) {	[22] REGMAP32 a2 =
[23] return m.Insert(k, v);	[23] return m.Insert(k, v);	[23] m.Insert2Kill(k);
[24] }	[24] }	[24] return a2.Insert2Use(v);
[25] ...	[25] ...	[25] }
[26]};	[26]};	[26]};

iterated. On each round, CodeSurfer/x86 applies a sequence of analyses: VSA, DUA, and several others. VSA is the primary workhorse, and it is often desirable for the information acquired by VSA to influence the outcomes of other analysis phases.

We can use the paired-semantics mechanism to obtain desired *multi-phase interactions* among our generated analyzers—typically, by pairing the VSA interpretation with another interpretation. For instance, with DUA_INTERP alone, the information required to get abstract memory location(s) for addr is lost because the DUA basetype-operators ($+$ and $*$ on line 3 of Fig. A.9) just return the union of the arguments' *use* sets (e.g., see lines 6–9 of the third column of Tab. A.2. With the pairing of VSA_INTERP with DUA_INTERP (line 1 of Fig. A.8), DUA can use the abstract address computed for addr2 by VSA_INTERP (line 6 of Fig. A.8), which uses VSA_INTERP::Add and VSA_INTERP::Mult; the latter operators operate on a numeric abstract domain (rather than a set-based one).

```
[1]
[2]   // User-defined abstract syntax
[3]  reg32:  EAX() | EBX();
[4]  flag:  ZF() | SF();
[5]  operand32
[6]      :Indirect32(reg32 INT32)
[7]      | DirectReg32(reg32)
[8]      | Immediate32(INT32)
[9] ;
[10] instruction
[11]     :ADD32_32(operand32 operand32)
[12]     | MOV32_32(operand32 operand32)
[13]     ;
[14] state:State(MAP[INT32,INT8]  // memory
[15]             MAP[reg32,INT32] // registers
[16]             MAP[flag,BOOL]); // flags
[17]
[18] // User-defined functions
[19] INT32 interpOp32( state S, operand32 I ) {
[20]  with(S) (
[21]   State(mem,regs,flags):
[22]    with(srcOp) (
[23]     DirectReg32(r):  regs(r),
[24]     Indirect32(base, disp):
[25]      let b = regs(base);
[26]          addr = b + disp;
[27]      ( mem(addr) )
[28]     Immediate32(i):  i
[29]    )
[30]  )
[31] };
[32] state updateFlag32(state S, ...)  {...}
[33] state updateState32(state S, ...)  {...}
[34] state interpInstr(instruction I, state S) {
[35]  with(I) (
[36]   MOV32_32(dstOp, srcOp):
[37]    let srcVal = interpOp32(S, srcOp);
[38]    in ( updateState32( S, dstOp, srcVal ) ),
[39]   ADD32_32(dstOp, srcOp):
[40]    let dstV = interpOp32(S, dstOp);
[41]        srcV = interpOp32(S, srcOp);
[42]        res = dstV + srcV;
[43]        S2 = updateFlag(S, dstV, srcV, res);
[44]    in ( updateState32( S2, dstOp, res ) )
[45]  )
[46] }
[47]
```

Figure A.4 A TSL specification of a simplified IA32 concrete semantics; reserved types and function names are underlined.

```
[1]  template <class BT> class CIR {
[2]   class reg32 {...};
[3]   class EAX: public reg32 {...};
[4]   ...
[5]   class operand32 {...};
[6]   class Indirect32:  public operand32 {...};
[7]   ...
[8]   class instruction {...};
[9]   class ADD32_32:  public instruction {
[10]    enum  TSL_ID  id;
[11]    operand32  op1;
[12]    operand32  op2;
[13]    ...
[14]  };
[15]  ...
[16]  class state# {...};
[17]  class State#:  public state# {...};
[18]  // User-defined functions
[19]  BT::INT32 interpOp32#( state# S, operand32 I ) {
[20]   with(S) (
[21]    State#(mem,regs,flags):
[22]     with(srcOp) (
[23]      DirectReg32(r):  BT::Access(regs,r),
[24]      Indirect32(base, disp):
[25]       let b = BT::Access(regs,base);
[26]           addr = BT::Plus(b, disp);
[27]       ( BT::Access(mem,addr) )
[28]      Immediate32(i):  i
[29]     )
[30]   )
[31]  };
[32]  state# updateFlag32#(state# S, ...)  {...}
[33]  state# updateState32#(state# S, ...)  {...}
[34]  state# interpInstr#(instruction I, state# S) {
[35]   with(I) (
[36]    MOV32_32(dstOp, srcOp):
[37]     let srcVal = interpOp32#(S, srcOp);
[38]     in ( updateState32#( S, dstOp, srcVal ) ),
[39]    ADD32_32(dstOp, srcOp):
[40]     let dstV = interpOp32#(S, dstOp);
[41]         srcV = interpOp32#(S, srcOp);
[42]         res = BT::Plus(dstV, srcV);
[43]         S2 = updateFlag#(S, dstV, srcV, res);
[44]     in ( updateState32#( S2, dstOp, res ) )
[45]   )
[46]  }
[47]};
```

Figure A.5 The CIR generated from Fig. A.4. (The superscript # is used to abbreviate the actual generated names used in the TSL implementation.)

Basetypes	Map-basetypes
constructors that handles concrete basetypes	constructors that handles concrete map-basetypes
bool approximates(const T & a)	bool approximates(const T & a)
T join(const T & a, const T & b)	T join(const T & a, const T & b)
T widen(const T & a, const T & b)	T widen(const T & a, const T & b)
T meet(const T & a, const T & b)	T meet(const T & a, const T & b)
bool isBottom()	bool isBottom()
void setToBottom()	void setToBottom()
static T BTM()	static T BTM()
bool isTop()	bool isTop()
void setToTop()	void setToTop()
static T TOP()	static T TOP()
bool operator ==	bool operator ==
bool operator $>$	TVL::Bool isEqual(const T & a)
TVL::Bool isEqual(const T & a)	std::ostream & print(std::ostream & o)
std::ostream & print(std::ostream & o)	T mapUpdate(const T & m, const K_T & key)
	D_T mapAccess(const T & m)

Figure A.6 Required operators that an abstract domain must provide for a TSL reinterpretation; T: the abstract domain for a maptype; K_T: the key type of the map-type T; D_T: the datum type of the map-type T; TVL::Bool: a three value logic (FALSE, ONE, and MAYBE);

```
[1] template <typename INTERP1, typename INTERP2>
[2] class PairedSemantics {
[3]   typedef PairedBaseType<INTERP1::INT32, INTERP2::INT32> INT32;
[4]   ...
[5]   INT32 MemAccess_32_8_LE_32(MEMMAP32_8_LE mem, INT32 addr) {
[6]     return INT32(INTERP1::MemAccess_32_8_LE_32(mem.GetFirst(), addr.GetFirst()),
[7]                  INTERP2::MemAccess_32_8_LE_32(mem.GetSecond(), addr.GetSecond()));
[8]   }
[9] };
```

Figure A.7 A part of the template class for paired semantics.

```
[1] typedef PairedSemantics<VSA_INTERP, DUA_INTERP> DUA;
[2] template<> DUA::INT32 DUA::MemAccess_32_8_LE_32( DUA::MEMMAP32_8_LE mem, DUA::INT32 addr) {
[3]    DUA::INTERP1::MEMMAP32_8_LE memory1 = mem.GetFirst();
[4]    DUA::INTERP2::MEMMAP32_8_LE memory2 = mem.GetSecond();
[5]    DUA::INTERP1::INT32 addr1 = addr.GetFirst();
[6]    DUA::INTERP2::INT32 addr2 = addr.GetSecond();
[7]    DUA::INT32 answer = interact(mem1, mem2, addr1, addr2);
[8]    return answer;
[9] }
```

Figure A.8 An example of C++ explicit template specialization to create a reduced product.

```
[1] with(op) ( ...
[2]  Indirect32(base, index, scale, disp):
[3]   let  addr = base + index * SignExtend8To32(scale) + disp;
[4]        m = MemUpdate_32_8_LE_32(mem,addr,v);
[5] ...)
```

Figure A.9 A fragment of updateState32.

Appendix B: Semantic-Reinterpretation for Symbolic-Analysis Primitives

In this appendix, we give correctness proofs for our generated primitives for symbolic evaluation, $\mathcal{WLP}$, and symbolic composition. These apply to the language PL (§4.2.2) and reinterpretations given in §4.3; the proofs for MC differ only slightly.

As a notational convenience, we do not distinguish between a *State* and a *LogicalStruct*. A *LogicalStruct* ι corresponds to the *State*: $((\iota\uparrow 1), (\iota\uparrow 2)F_\rho)$. Because, for PL, logical structures only contain the single function F_ρ, there is a one-to-one correspondence with states. Hence, whenever necessary (e.g. in the applications of $\mathcal{E}[\![.]\!]$, $\mathcal{B}[\![.]\!]$, and $\mathcal{I}[\![.]\!]$), we assume that that a *LogicalStruct* ι is coerced to $((\iota\uparrow 1), (\iota\uparrow 2)F_\rho)$.

B.1 Correctness of the Symbolic-Evaluation Primitive

Lemma B.1 (Relationship of $\overline{\mathcal{E}}$ to $\mathcal{E}$ and $\overline{\mathcal{B}}$ to $\mathcal{B}$)

$$(1)\ \mathcal{T}[\![\overline{\mathcal{E}}[\![E]\!]U]\!]\iota = \mathcal{E}[\![E]\!](\mathcal{U}[\![U]\!]\iota)$$
$$(2)\ \mathcal{F}[\![\overline{\mathcal{B}}[\![BE]\!]U]\!]\iota = \mathcal{B}[\![BE]\!](\mathcal{U}[\![U]\!]\iota)$$

Proof: The two lemmas are simultaneously proved using structural induction on E and BE, as shown below. Let U be $(\{I_i \hookleftarrow T_i\}, \{F_j \hookleftarrow FE_j\})$.

Note that the standard interpretations of *binop*, *relop*, and *boolop* coincide with those of $binop_L$, $relop_L$, and $boolop_L$. Thus, reasoning steps of the form $binop_L(op2_L) \rightsquigarrow binop(op2)$ are shorthands for reasoning about each case, such as $binop_L(\boxed{+}) \rightsquigarrow binop(+)$, etc.

$(1)\ (i)$

$$\begin{aligned}\mathcal{T}[\![\overline{\mathcal{E}}[\![c]\!]U]\!]\iota &= \mathcal{T}[\![\overline{const}(c)]\!]\iota \\ &= \mathcal{T}[\![c]\!]\iota \\ &= const(c) \\ &= \mathcal{E}[\![c]\!](\mathcal{U}[\![U]\!]\iota)\end{aligned}$$

(ii)

lhs : $\mathcal{T}[\![\overline{\mathcal{E}}[\![I]\!]U]\!]\iota$

$= \mathcal{T}[\![\overline{\textit{lookupState}}\, U\ I]\!]\iota$

$= \mathcal{T}[\![((U{\uparrow}2)F_\rho)((U{\uparrow}1)I)]\!]\iota$

rhs : $\mathcal{E}[\![I]\!](\mathcal{U}[\![U]\!]\iota)$

$= \mathcal{E}[\![I]\!] \begin{pmatrix} (\iota{\uparrow}1)[I_i \mapsto \mathcal{T}[\![(U{\uparrow}1)I_i]\!]\iota], \\ (\iota{\uparrow}2)[F_j \mapsto \mathcal{FE}[\![(U{\uparrow}2)F_j]\!]\iota) \end{pmatrix}$

$= \textit{lookupState} \begin{pmatrix} (\iota{\uparrow}1)[I_i \mapsto \mathcal{T}[\![(U{\uparrow}1)I_i]\!]\iota], \\ (\iota{\uparrow}2)[F_j \mapsto \mathcal{FE}[\![(U{\uparrow}2)F_j]\!]\iota)I \end{pmatrix}$

$= ((\iota{\uparrow}2)[F_j \mapsto \mathcal{FE}[\![(U{\uparrow}2)F_j]\!]\iota)\ (\mathcal{T}[\![(U{\uparrow}1)I]\!]\iota)$

$= (\mathcal{FE}[\![(U{\uparrow}2)F_\rho]\!]\iota)\ (\mathcal{T}[\![(U{\uparrow}1)I]\!]\iota)$

$= \textit{access}(\mathcal{FE}[\![(U{\uparrow}2)F_\rho]\!]\iota, \mathcal{T}[\![(U{\uparrow}1)I]\!]\iota)$

$= \mathcal{T}[\![((U{\uparrow}2)F_\rho)((U{\uparrow}1)I)]\!]\iota$

(iii)

lhs : $\mathcal{T}[\![\overline{\mathcal{E}}[\![\&I]\!]U]\!]\iota = \mathcal{T}[\![\overline{\textit{lookupEnv}}\, U\ I]\!]\iota = \mathcal{T}[\![(U{\uparrow}1)I]\!]\iota$

rhs : $\mathcal{E}[\![\&I]\!](\mathcal{U}[\![U]\!]\iota)$

$= \mathcal{E}[\![\&I]\!] \begin{pmatrix} (\iota{\uparrow}1)[I_i \mapsto \mathcal{T}[\![(U{\uparrow}1)I_i]\!]\iota], \\ (\iota{\uparrow}2)[F_j \mapsto \mathcal{FE}[\![(U{\uparrow}2)F_j]\!]\iota) \end{pmatrix}$

$= \textit{lookupEnv} \begin{pmatrix} (\iota{\uparrow}1)[I_i \mapsto \mathcal{T}[\![(U{\uparrow}1)I_i]\!]\iota], \\ (\iota{\uparrow}2)[F_j \mapsto \mathcal{FE}[\![(U{\uparrow}2)F_j]\!]\iota)I \end{pmatrix}$

$= \mathcal{T}[\![(U{\uparrow}1)I]\!]\iota$

(iv)

lhs :

$$= \mathcal{T}[\![\overline{\mathcal{E}}[\![*E]\!]U]\!]\iota$$
$$= \mathcal{T}[\![\overline{\textit{lookupStore}}\ U\ (\overline{\mathcal{E}}[\![E]\!]U)]\!]\iota$$
$$= \mathcal{T}[\![((U{\uparrow}2)F_\rho)(\overline{\mathcal{E}}[\![E]\!]U)]\!]\iota$$

rhs : $\mathcal{E}[\![*E]\!](\mathcal{U}[\![U]\!]\iota)$

$$= \mathcal{E}[\![*E]\!]\begin{pmatrix}(\iota{\uparrow}1)[I_i \mapsto \mathcal{T}[\![(U{\uparrow}1)I_i]\!]\iota],\\ (\iota{\uparrow}2)[F_j \mapsto \mathcal{FE}[\![(U{\uparrow}2)F_j]\!]\iota)\end{pmatrix}$$
$$= \textit{lookupStore}\begin{pmatrix}(\iota{\uparrow}1)[I_i \mapsto \mathcal{T}[\![(U{\uparrow}1)I_i]\!]\iota],\\ (\iota{\uparrow}2)[F_j \mapsto \mathcal{FE}[\![(U{\uparrow}2)F_j]\!]\iota)(\mathcal{E}[\![E]\!](\mathcal{U}[\![U]\!]\iota))\end{pmatrix}$$
$$= (\mathcal{FE}[\![(U{\uparrow}2)F_\rho]\!]\iota)\ (\mathcal{E}[\![E]\!](\mathcal{U}[\![U]\!]\iota))$$
$$= (\mathcal{FE}[\![(U{\uparrow}2)F_\rho]\!]\iota)\ (\mathcal{T}[\![\overline{\mathcal{E}}[\![E]\!]U]\!]\iota)\ //\ \text{by ind. via } (1)$$
$$= \textit{access}(\mathcal{FE}[\![(U{\uparrow}2)F_\rho]\!]\iota, \mathcal{T}[\![\overline{\mathcal{E}}[\![E]\!]U]\!]\iota)$$
$$= \mathcal{T}[\![((U{\uparrow}2)F_\rho)(\overline{\mathcal{E}}[\![E]\!]U)]\!]\iota$$

$(v)\ \mathcal{T}[\![\overline{\mathcal{E}}[\![E_1 \textit{op2}\ E_2]\!]U]\!]\iota$

$$= \mathcal{T}[\![\overline{\mathcal{E}}[\![E_1]\!]U\ \textit{op2}_L\ \overline{\mathcal{E}}[\![E_2]\!]U]\!]\iota$$
$$= \mathcal{T}[\![\overline{\mathcal{E}}[\![E_1]\!]U]\!]\iota\ \textit{binop}_L(\textit{op2}_L)\ \mathcal{T}[\![\overline{\mathcal{E}}[\![E_2]\!]U]\!]\iota$$
$$= \mathcal{E}[\![E_1]\!](\mathcal{U}[\![U]\!]\iota)\ \textit{binop}(\textit{op2})\ \mathcal{E}[\![E_2]\!](\mathcal{U}[\![U]\!]\iota)$$
$$//\ \text{by ind. via } (1)$$
$$= \mathcal{E}[\![E_1\ \textit{op2}\ E_2]\!](\mathcal{U}[\![U]\!]\iota)$$

$(vi)\ \mathcal{T}[\![\overline{\mathcal{E}}[\![\textit{BE}\ ?\ E_1 : E_2]\!]U]\!]\iota$

$$= \mathcal{T}[\![\textit{ite}(\overline{\mathcal{B}}[\![\textit{BE}]\!]U, \overline{\mathcal{E}}[\![E_1]\!]U, \overline{\mathcal{E}}[\![E_2]\!]U)]\!]\iota$$
$$= \textit{cond}_L(\mathcal{F}[\![\overline{\mathcal{B}}[\![\textit{BE}]\!]U]\!]\iota, \mathcal{T}[\![\overline{\mathcal{E}}[\![E_1]\!]U]\!]\iota, \mathcal{T}[\![\overline{\mathcal{E}}[\![E_2]\!]U]\!]\iota)$$
$$= \mathcal{F}[\![\overline{\mathcal{B}}[\![\textit{BE}]\!]U]\!]\iota\ ?\ \mathcal{T}[\![\overline{\mathcal{E}}[\![E_1]\!]U]\!]\iota : \mathcal{T}[\![\overline{\mathcal{E}}[\![E_2]\!]U]\!]\iota$$
$$= \mathcal{B}[\![\textit{BE}]\!](\mathcal{U}[\![U]\!]\iota)\ ?\ \mathcal{E}[\![E_1]\!](\mathcal{U}[\![U]\!]\iota) : \mathcal{E}[\![E_2]\!](\mathcal{U}[\![U]\!]\iota)$$
$$//\ \text{by ind. via } (1) \text{ and } (2)$$
$$= \mathcal{E}[\![\textit{BE}\ ?\ E_1 : E_2]\!](\mathcal{U}[\![U]\!]\iota)$$

$(2)\ (i)\ \mathcal{F}[\![\overline{\mathcal{B}}[\![\mathbb{T}]\!]U]\!]\iota = \mathcal{F}[\![\mathbb{T}]\!]\iota = \mathbb{T} = \mathcal{B}[\![\mathbb{T}]\!](\mathcal{U}[\![U]\!]\iota)$

(ii) $\mathcal{F}[\![\overline{\mathcal{B}}[\![\mathbb{F}]\!]U]\!]\iota = \mathcal{F}[\![\mathbb{F}]\!]\iota = \mathbb{F} = \mathcal{B}[\![\mathbb{F}]\!](\mathcal{U}[\![U]\!]\iota)$

(iii) $\mathcal{F}[\![\overline{\mathcal{B}}[\![E_1 \ \mathit{rop}\ E_2]\!]U]\!]\iota$

$= \mathcal{F}[\![\overline{\mathcal{E}}[\![E_1]\!]U \ \mathit{rop}_L\ \overline{\mathcal{E}}[\![E_2]\!]U]\!]\iota$

$= \mathcal{T}[\![\overline{\mathcal{E}}[\![E_1]\!]U]\!]\iota \ \mathit{relop}_L(\mathit{rop}_L)\ \mathcal{T}[\![\overline{\mathcal{E}}[\![E_2]\!]U]\!]\iota$

$= \mathcal{E}[\![E_1]\!](\mathcal{U}[\![U]\!]\iota) \ \mathit{relop}(\mathit{rop})\ \mathcal{E}[\![E_2]\!](\mathcal{U}[\![U]\!]\iota)$

// by ind. via (1)

$= \mathcal{B}[\![E_1 \ \mathit{rop}\ E_2]\!](\mathcal{U}[\![U]\!]\iota)$

(iv) $\mathcal{F}[\![\overline{\mathcal{B}}[\![\neg \mathit{BE}_1]\!]U]\!]\iota$

$= \mathcal{F}[\![\boxed{\neg}\,\overline{\mathcal{B}}[\![\mathit{BE}_1]\!]U]\!]\iota$

$= \neg\mathcal{F}[\![\overline{\mathcal{B}}[\![\mathit{BE}_1]\!]U]\!]\iota$

$= \neg\mathcal{B}[\![\mathit{BE}_1]\!](\mathcal{U}[\![U]\!]\iota)$ // by ind. via (2)

$= \mathcal{B}[\![\neg \mathit{BE}_1]\!](\mathcal{U}[\![U]\!]\iota)$

(v) $\mathcal{F}[\![\overline{\mathcal{B}}[\![\mathit{BE}_1 \ \mathit{bop}\ \mathit{BE}_2]\!]U]\!]\iota$

$= \mathcal{F}[\![\overline{\mathcal{B}}[\![\mathit{BE}_1]\!]U \ \mathit{bop}_L\ \overline{\mathcal{B}}[\![\mathit{BE}_2]\!]U]\!]\iota$

$= \mathcal{F}[\![\overline{\mathcal{B}}[\![\mathit{BE}_1]\!]U]\!]\iota \ \mathit{boolop}_L(\mathit{bop}_L)\ \mathcal{F}[\![\overline{\mathcal{B}}[\![\mathit{BE}_2]\!]U]\!]\iota$

$= \mathcal{B}[\![\mathit{BE}_1]\!](\mathcal{U}[\![U]\!]\iota) \ \mathit{boolop}(\mathit{bop})\ \mathcal{B}[\![\mathit{BE}_2]\!](\mathcal{U}[\![U]\!]\iota)$

// by ind. via (2)

$= \mathcal{B}[\![\mathit{BE}_1 \ \mathit{bop}\ \mathit{BE}_2]\!](\mathcal{U}[\![U]\!]\iota)$

□

Theorem 4.4 *For all $s \in$ Stmt, $U \in$ StructUpdate, and $\iota \in$ LogicalStruct, the meaning of $\overline{\mathcal{I}}[\![s]\!]U$ in ι (i.e., $\mathcal{U}[\![\overline{\mathcal{I}}[\![s]\!]U]\!]\iota$) is equivalent to running $\mathcal{I}$ on s with an input state obtained from $\mathcal{U}[\![U]\!]\iota$. That is,*

$$\mathcal{U}[\![\overline{\mathcal{I}}[\![s]\!]U]\!]\iota = \mathcal{I}[\![s]\!](\mathcal{U}[\![U]\!]\iota).$$

□

Proof:

$$
\begin{aligned}
(i)\ &\mathcal{U}[\![\overline{\mathcal{I}}[\![I = E;]\!]U]\!]\iota \\
&= \mathcal{U}[\![\overline{\textit{updateStore}}\ U\ (\overline{\textit{lookupEnv}}\ U\ I)\ (\overline{\mathcal{E}}[\![E]\!]U)]\!]\iota \\
&= \mathcal{U}[\![\overline{\textit{updateStore}}\ U\ ((U{\uparrow}1)I)\ (\overline{\mathcal{E}}[\![E]\!]U)]\!]\iota \\
&= \mathcal{U}\left[\!\!\left[\begin{pmatrix}(U{\uparrow}1),\\ (U{\uparrow}2)[F_\rho \mapsto ((U{\uparrow}2)F_\rho)[(U{\uparrow}1)I \mapsto \overline{\mathcal{E}}[\![E]\!]U]])\end{pmatrix}\right]\!\!\right]\iota \\
&= \begin{pmatrix}(\mathcal{U}[\![U]\!]\iota{\uparrow}1),\\ (\mathcal{U}[\![U]\!]\iota{\uparrow}2)[\mathcal{T}[\![(\mathcal{U}[\![U]\!]\iota)I]\!] \mapsto \mathcal{T}[\![\overline{\mathcal{E}}[\![E]\!](\mathcal{U}[\![U]\!]\iota)]\!]]\end{pmatrix} \\
&= \begin{pmatrix}(\mathcal{U}[\![U]\!]\iota{\uparrow}1),\\ (\mathcal{U}[\![U]\!]\iota{\uparrow}2)[(\mathcal{U}[\![U]\!]\iota{\uparrow}1)I \mapsto \mathcal{E}[\![E]\!](\mathcal{U}[\![U]\!]\iota)]\end{pmatrix} \\
&\qquad\qquad // \text{ by Lem. B.1(1)} \\
&= \textit{updateStore}\ \ (\mathcal{U}[\![U]\!]\iota) \\
&\qquad\qquad (\textit{lookupEnv}\ (\mathcal{U}[\![U]\!]\iota)\ I) \\
&\qquad\qquad (\mathcal{E}[\![E]\!](\mathcal{U}[\![U]\!]\iota)) \\
&= \mathcal{I}[\![I = E;]\!](\mathcal{U}[\![U]\!]\iota)
\end{aligned}
$$

$$
\begin{aligned}
(ii)\ &\mathcal{U}[\![\overline{\mathcal{I}}[\![*I = E;]\!]U]\!]\iota \\
&= \mathcal{U}[\![\overline{\textit{updateStore}}\ U\ (\overline{\mathcal{E}}[\![I]\!]U)\ (\overline{\mathcal{E}}[\![E]\!]U)]\!]\iota \\
&= \mathcal{U}\left[\!\!\left[\begin{pmatrix}(U{\uparrow}1),\\ (U{\uparrow}2)[F_\rho \mapsto ((U{\uparrow}2)F_\rho)[\overline{\mathcal{E}}[\![I]\!]U \mapsto \overline{\mathcal{E}}[\![E]\!]U]]\end{pmatrix}\right]\!\!\right]\iota \\
&= \begin{pmatrix}(\mathcal{U}[\![U]\!]\iota{\uparrow}1),\\ (\mathcal{U}[\![U]\!]\iota{\uparrow}2)[\mathcal{T}[\![\overline{\mathcal{E}}[\![I]\!](\mathcal{U}[\![U]\!]\iota)]\!] \mapsto \mathcal{T}[\![\overline{\mathcal{E}}[\![E]\!](\mathcal{U}[\![U]\!]\iota)]\!]]\end{pmatrix} \\
&= \begin{pmatrix}(\mathcal{U}[\![U]\!]\iota{\uparrow}1),\\ (\mathcal{U}[\![U]\!]\iota{\uparrow}2)[\mathcal{E}[\![I]\!](\mathcal{U}[\![U]\!]\iota) \mapsto \mathcal{E}[\![E]\!](\mathcal{U}[\![U]\!]\iota)]\end{pmatrix} \\
&\qquad\qquad // \text{ by Lem. B.1(1)} \\
&= \textit{updateStore}\ (\mathcal{U}[\![U]\!]\iota)\ (\mathcal{E}[\![I]\!](\mathcal{U}[\![U]\!]\iota))\ (\mathcal{E}[\![E]\!](\mathcal{U}[\![U]\!]\iota)) \\
&= \mathcal{I}[\![*I = E;]\!](\mathcal{U}[\![U]\!]\iota)
\end{aligned}
$$

(iii) $(\mathcal{U}[\![\overline{\mathcal{I}}[\![S_1 S_2]\!]U]\!]\iota)$

$= (\mathcal{U}[\![\overline{\mathcal{I}}[\![S_2]\!](\overline{\mathcal{I}}[\![S_1]\!]U)]\!]\iota)$

$= \mathcal{I}[\![S_2]\!](\mathcal{U}[\![\overline{\mathcal{I}}[\![S_1]\!]U]\!]\iota)$ // by induction

$= \mathcal{I}[\![S_2]\!](\mathcal{I}[\![S_1]\!](\mathcal{U}[\![U]\!]\iota))$ // by induction

$= \mathcal{I}[\![S_1 S_2]\!](\mathcal{U}[\![U]\!]\iota)$ □

B.2 Correctness of $\mathcal{WLP}$

Lemma B.2 (Relationship of $\overline{\mathcal{T}}$ to $\mathcal{T}$, $\overline{\mathcal{F}}$ to $\mathcal{F}$, $\overline{\mathcal{FE}}$ to $\mathcal{FE}$)

$$(1)\ \mathcal{T}[\![\overline{\mathcal{T}}[\![T]\!]U]\!]\iota = \mathcal{T}[\![T]\!](\mathcal{U}[\![U]\!]\iota)$$
$$(2)\ \mathcal{F}[\![\overline{\mathcal{F}}[\![\varphi]\!]U]\!]\iota = \mathcal{F}[\![\varphi]\!](\mathcal{U}[\![U]\!]\iota)$$
$$(3)\ \mathcal{FE}[\![\overline{\mathcal{FE}}[\![FE]\!]U]\!]\iota = \mathcal{FE}[\![FE]\!](\mathcal{U}[\![U]\!]\iota)$$

Proof: The three lemmas are simultaneously proved using structural induction on T, φ, and *FE*, as shown below. Let U be $(\{I_i \hookleftarrow T_i\}, \{F_j \hookleftarrow FE_j\})$. (Thus, $T_i = (U{\uparrow}1)I_i$ and $FE_j = (U{\uparrow}2)F_j$.) Let f be $(\iota{\uparrow}2)[F_j \mapsto \mathcal{FE}[\![FE_j]\!]\iota]$.

(1) (i) $\mathcal{T}[\![\overline{\mathcal{T}}[\![c]\!]U]\!]\iota = \mathcal{T}[\![c]\!]\iota = \mathit{const}(c) = \mathcal{T}[\![c]\!](\mathcal{U}[\![U]\!]\iota)$

(ii)

lhs $= \mathcal{T}[\![\overline{\mathcal{T}}[\![I]\!]U]\!]\iota = \mathcal{T}[\![\overline{\mathit{lookupId}}\ U\ I]\!]\iota = \mathcal{T}[\![(U{\uparrow}1)I]\!]\iota$

rhs $= \mathcal{T}[\![I]\!](\mathcal{U}[\![U]\!]\iota) = \mathcal{T}[\![I]\!]((\iota{\uparrow}1)[I_i \mapsto \mathcal{T}[\![T_i]\!]\iota], f)$

$= \mathit{lookupId}\ ((\iota{\uparrow}1)[I_i \mapsto \mathcal{T}[\![T_i]\!]\iota], f)\ I$

$= \mathcal{T}[\![(U{\uparrow}1)I]\!]\iota$

(iii) $\mathcal{T}[\![\overline{\mathcal{T}}[\![T_1\ \mathit{op2}_L\ T_2]\!]U]\!]\iota$

$= \mathcal{T}[\![\overline{\mathcal{T}}[\![T_1]\!]U\ \mathit{op2}_L\ \overline{\mathcal{T}}[\![T_2]\!]U]\!]\iota$

$= \mathcal{T}[\![\overline{\mathcal{T}}[\![T_1]\!]U]\!]\iota\ \mathit{binop}_L(\mathit{op2}_L)\ \mathcal{T}[\![\overline{\mathcal{T}}[\![T_2]\!]U]\!]\iota$

$= \mathcal{T}[\![T_1]\!](\mathcal{U}[\![U]\!]\iota)\ \mathit{binop}_L(\mathit{op2}_L)\ \mathcal{T}[\![T_2]\!](\mathcal{U}[\![U]\!]\iota)$

// by ind. via (1)

$= \mathcal{T}[\![T_1\ \mathit{op2}_L\ T_2]\!](\mathcal{U}[\![U]\!]\iota)$

$(iv)\ \mathcal{T}[\![\overline{\mathcal{T}}[\![\mathit{ite}(\varphi, T_1, T_2)]\!]U]\!]\iota$

$= \mathcal{T}[\![\mathit{ite}(\overline{\mathcal{F}}[\![\varphi]\!]U, \overline{\mathcal{T}}[\![T_1]\!]U, \overline{\mathcal{T}}[\![T_2]\!]U)]\!]\iota$

$= \mathit{cond}_L(\mathcal{F}[\![\overline{\mathcal{F}}[\![\varphi]\!]U]\!]\iota, \mathcal{T}[\![\overline{\mathcal{T}}[\![T_1]\!]U]\!]\iota, \mathcal{T}[\![\overline{\mathcal{T}}[\![T_2]\!]U]\!]\iota)$

$= \mathcal{F}[\![\overline{\mathcal{F}}[\![\varphi]\!]U]\!]\iota\ ?\ \mathcal{T}[\![\overline{\mathcal{T}}[\![T_1]\!]U]\!]\iota : \mathcal{T}[\![\overline{\mathcal{T}}[\![T_2]\!]U]\!]\iota$

$= \mathcal{F}[\![\varphi]\!](\mathcal{U}[\![U]\!]\iota)\ ?\ \mathcal{T}[\![T_1]\!](\mathcal{U}[\![U]\!]\iota) : \mathcal{T}[\![T_2]\!](\mathcal{U}[\![U]\!]\iota)$

// by ind. via (1) and (2)

$= \mathcal{F}[\![\varphi\ ?\ T_1 : T_2]\!](\mathcal{U}[\![U]\!]\iota)$

$(v)\ \mathcal{T}[\![\overline{\mathcal{T}}[\![\mathit{FE}(T)]\!]U]\!]\iota$

$= \mathcal{T}[\![\overline{\mathcal{F}\mathcal{E}}[\![\mathit{FE}]\!]U(\overline{\mathcal{T}}[\![T]\!]U)]\!]\iota$

$= (\mathcal{F}\mathcal{E}[\![\overline{\mathcal{F}\mathcal{E}}[\![\mathit{FE}]\!]U]\!]\iota)(\mathcal{T}[\![\overline{\mathcal{T}}[\![T]\!]U]\!]\iota)$

$= (\mathcal{F}\mathcal{E}[\![\mathit{FE}]\!](\mathcal{U}[\![U]\!]\iota))(\mathcal{T}[\![T]\!](\mathcal{U}[\![U]\!]\iota))$

// by ind. via (3)

$= \mathcal{T}[\![\mathit{FE}(T)]\!](\mathcal{U}[\![U]\!]\iota)$

$(2)\ (i)\ \mathcal{F}[\![\overline{\mathcal{F}}[\![\boxed{\mathbb{T}}]\!]U]\!]\iota = \mathcal{F}[\![\boxed{\mathbb{T}}]\!]\iota = \mathbb{T} = \mathcal{F}[\![\boxed{\mathbb{T}}]\!](\mathcal{U}[\![U]\!]\iota)$

$(ii)\ \mathcal{F}[\![\overline{\mathcal{F}}[\![\boxed{\mathbb{F}}]\!]U]\!]\iota = \mathcal{F}[\![\boxed{\mathbb{F}}]\!]\iota = \mathbb{F} = \mathcal{F}[\![\boxed{\mathbb{F}}]\!](\mathcal{U}[\![U]\!]\iota)$

$(iii)\ \mathcal{F}[\![\overline{\mathcal{F}}[\![T_1\ \mathit{rop}_L\ T_2]\!]U]\!]\iota$

$= \mathcal{F}[\![\overline{\mathcal{T}}[\![T_1]\!]U\ \mathit{relop}_L(\mathit{rop}_L)\ \overline{\mathcal{T}}[\![T_2]\!]U]\!]\iota$

$= \mathcal{T}[\![\overline{\mathcal{T}}[\![T_1]\!]U]\!]\iota\ \mathit{relop}_L(\mathit{rop}_L)\ \mathcal{T}[\![\overline{\mathcal{T}}[\![T_2]\!]U]\!]\iota$

$= \mathcal{T}[\![T_1]\!](\mathcal{U}[\![U]\!]\iota)\ \mathit{relop}_L(\mathit{rop}_L)\ \mathcal{T}[\![T_2]\!](\mathcal{U}[\![U]\!]\iota)$

// by ind. via (1)

$= \mathcal{F}[\![T_1\ \mathit{rop}_L\ T_2]\!](\mathcal{U}[\![U]\!]\iota)$

$(iv)\ \mathcal{F}[\![\overline{\mathcal{F}}[\![\boxed{\neg}\,\varphi_1]\!]U]\!]\iota$

$= \mathcal{F}[\![\boxed{\neg}\,\overline{\mathcal{F}}[\![\varphi_1]\!]U]\!]\iota$

$= \neg\mathcal{F}[\![\overline{\mathcal{F}}[\![\varphi_1]\!]U]\!]\iota$

$= \neg\mathcal{F}[\![\varphi_1]\!](\mathcal{U}[\![U]\!]\iota)$ // by ind. via (2)

$= \mathcal{F}[\![\boxed{\neg}\,\varphi_1]\!](\mathcal{U}[\![U]\!]\iota)$

$(v)\ \mathcal{F}[\![\overline{\mathcal{F}}[\![\varphi_1\ \mathit{bop}_L\ \varphi_2]\!]U]\!]\iota$

$= \mathcal{F}[\![\overline{\mathcal{F}}[\![\varphi_1]\!]U\ \mathit{boolop}_L(\mathit{bop}_L)\ \overline{\mathcal{F}}[\![\varphi_2]\!]U]\!]\iota$

$= \mathcal{F}[\![\overline{\mathcal{F}}[\![\varphi_1]\!]U]\!]\iota\ \mathit{boolop}_L(\mathit{bop}_L)\ \mathcal{F}[\![\overline{\mathcal{F}}[\![\varphi_2]\!]U]\!]\iota$

$= \mathcal{F}[\![\varphi_1]\!](\mathcal{U}[\![U]\!]\iota)\ \mathit{boolop}_L(\mathit{bop}_L)\ \mathcal{F}[\![\varphi_2]\!](\mathcal{U}[\![U]\!]\iota)$

// by ind. via (2)

$= \mathcal{F}[\![\varphi_1\ \mathit{bop}_L\ \varphi_2]\!](\mathcal{U}[\![U]\!]\iota)$

$(3)\ (i)$

lhs $= \mathcal{FE}[\![\overline{\mathcal{FE}}[\![F]\!]U]\!]\iota$

$= \mathcal{FE}[\![\overline{\mathit{lookupId}}\ U\ F]\!]\iota$

$= \mathcal{FE}[\![(U{\uparrow}2)F]\!]\iota$

rhs $= \mathcal{FE}[\![F]\!](\mathcal{U}[\![U]\!]\iota)$

$= \mathcal{FE}[\![F]\!]((\iota{\uparrow}1)[I_i \mapsto \mathcal{T}[\![T_i]\!]\iota], f)$

$= \mathit{lookupFuncId}\ ((\iota{\uparrow}1)[I_i \mapsto \mathcal{T}[\![T_i]\!]\iota], f)\ F$

$= \mathcal{FE}[\![(U{\uparrow}2)F]\!]\iota$

$(ii)\ \mathcal{FE}[\![\overline{\mathcal{FE}}[\![\mathit{FE}_0[T_1 \mapsto T_2]]\!]U]\!]\iota$

$= \mathcal{FE}[\![(\overline{\mathcal{FE}}[\![\mathit{FE}_0]\!]U)[\overline{\mathcal{T}}[\![T_1]\!]U \mapsto \overline{\mathcal{T}}[\![T_2]\!]U]]\!]\iota$

$= \mathcal{FE}[\![(\overline{\mathcal{FE}}[\![\mathit{FE}_0]\!]U)]\!]\iota[\mathcal{T}[\![\overline{\mathcal{T}}[\![T_1]\!]U]\!]\iota \mapsto \mathcal{T}[\![\overline{\mathcal{T}}[\![T_2]\!]U]\!]\iota]$

$= \mathcal{FE}[\![\mathit{FE}_0]\!](\mathcal{U}[\![U]\!]\iota)[\mathcal{T}[\![T_1]\!](\mathcal{U}[\![U]\!]\iota) \mapsto \mathcal{T}[\![T_2]\!](\mathcal{U}[\![U]\!]\iota)]$

// by ind. via (1)

$= \mathcal{FE}[\![\mathit{FE}_0[T_1 \mapsto T_2]]\!](\mathcal{U}[\![U]\!]\iota)$

□

Theorem 4.9 *For any Stmt s and Formula φ, $\psi := \overline{\mathcal{F}}[\![\varphi]\!](\overline{\mathcal{I}}[\![s]\!]U_{id})$ is an acceptable $\mathcal{WLP}$ formula for φ with respect to s.* □

Proof: For all $\iota \in$ *LogicalStruct*,

$$\begin{aligned}\mathcal{F}[\![\psi]\!]\iota &= \mathcal{F}[\![\overline{\mathcal{F}}[\![\varphi]\!](\overline{\mathcal{I}}[\![s]\!]U_{id})]\!]\iota \\ &= \mathcal{F}[\![\varphi]\!](\mathcal{U}[\![\overline{\mathcal{I}}[\![s]\!]U_{id}]\!]\iota) \quad \text{// by Lem. B.2} \\ &= \mathcal{F}[\![\varphi]\!](\mathcal{I}[\![s]\!](\mathcal{U}[\![U_{id}]\!]\iota)) \quad \text{// by Thm. 4.4} \\ &= \mathcal{F}[\![\varphi]\!](\mathcal{I}[\![s]\!]\iota)\end{aligned}$$

and therefore, by Defn. 4.8, $\overline{\mathcal{F}}[\![\varphi]\!](\overline{\mathcal{I}}[\![s]\!]U_{id})$ is an acceptable $\mathcal{WLP}$ formula for φ with respect to s. □

B.3 Correctness of the Symbolic-Composition Primitive

We now show that the meaning of $\overline{\mathcal{U}}[\![U_2]\!]U_1$ is the composition of the meanings of U_2 and U_1.

Theorem 4.11 *For all $U_1, U_2 \in$ StructUpdate,*

$$\mathcal{U}[\![\overline{\mathcal{U}}[\![U_2]\!]U_1]\!] = \mathcal{U}[\![U_2]\!] \circ \mathcal{U}[\![U_1]\!].$$

□ **Proof:** Let $U_2 = (\{I_i \hookleftarrow T_i\}, \{F_j \hookleftarrow FE_j\})$; let I_k and F_m range over *Id* and *FuncId*, respectively; and let $\iota \in$ *LogicalStruct* be an arbitrary logical structure.

$$\begin{aligned}
&\mathcal{U}[\![\overline{\mathcal{U}}[\![U_2]\!]U_1]\!]\iota \\
&= \mathcal{U}\left[\!\!\left[\begin{pmatrix}(U_1{\uparrow}1)[I_i \mapsto \overline{\mathcal{T}}[\![T_i]\!]U_1], \\ (U_1{\uparrow}2)[F_j \mapsto \overline{\mathcal{FE}}[\![FE_j]\!]U_1]\end{pmatrix}\right]\!\!\right]\iota \\
&= \mathcal{U}\left[\!\!\left[\begin{pmatrix}\{I_k \mapsto ((U_1{\uparrow}1)[I_i \mapsto \overline{\mathcal{T}}[\![T_i]\!]U_1])I_k\}, \\ \{F_m \mapsto ((U_1{\uparrow}2)[F_j \mapsto \overline{\mathcal{FE}}[\![FE_j]\!]U_1])F_m\}\end{pmatrix}\right]\!\!\right]\iota \\
&= \begin{pmatrix}(\iota{\uparrow}1)[I_k \mapsto \mathcal{T}[\![((U_1{\uparrow}1)[I_i \mapsto \overline{\mathcal{T}}[\![T_i]\!]U_1])I_k]\!]\iota], \\ (\iota{\uparrow}2)[F_m \mapsto \mathcal{FE}[\![((U_1{\uparrow}2)[F_j \mapsto \overline{\mathcal{FE}}[\![FE_j]\!]U_1])F_m]\!]\iota]\end{pmatrix} \\
&= \begin{pmatrix}(\iota{\uparrow}1)[I_{k(\neq i)} \mapsto \mathcal{T}[\![(U_1{\uparrow}1)I_k]\!]\iota][I_i \mapsto \mathcal{T}[\![\overline{\mathcal{T}}[\![T_i]\!]U_1]\!]\iota], \\ (\iota{\uparrow}2)[F_{m(\neq j)} \mapsto \mathcal{FE}[\![(U_1{\uparrow}2)F_m]\!]\iota][F_j \mapsto \mathcal{FE}[\![\overline{\mathcal{FE}}[\![FE_j]\!]U_1]]\!]\iota\end{pmatrix} \\
&= \begin{pmatrix}(\iota{\uparrow}1)[I_{k(\neq i)} \mapsto \mathcal{T}[\![(U_1{\uparrow}1)I_k]\!]\iota][I_i \mapsto \mathcal{T}[\![T_i]\!](\mathcal{U}[\![U_1]\!]\iota)], \\ (\iota{\uparrow}2)[F_{m(\neq j)} \mapsto \mathcal{FE}[\![(U_1{\uparrow}2)F_m]\!]\iota][F_j \mapsto \mathcal{FE}[\![FE_j]\!](\mathcal{U}[\![U_1]\!]\iota)]\end{pmatrix} \\
&\qquad\qquad // \text{ by Lem. B.2} \\
&= \begin{pmatrix}((\mathcal{U}[\![U_1]\!]\iota){\uparrow}1)[I_i \mapsto \mathcal{T}[\![T_i]\!](\mathcal{U}[\![U_1]\!]\iota)], \\ ((\mathcal{U}[\![U_1]\!]\iota){\uparrow}2)[F_j \mapsto \mathcal{FE}[\![FE_j]\!](\mathcal{U}[\![U_1]\!]\iota)]\end{pmatrix} \\
&= \mathcal{U}[\![U_2]\!](\mathcal{U}[\![U_1]\!]\iota)
\end{aligned}$$

□